REGION BASED IMAGE COMPOSITING

Field of the Invention

The present invention relates to the creation of computer-generated images both in the form of still pictures and video imagery, and, in particular, relates to efficient process, apparatus, and system for creating an image made up by compositing multiple components.

Background

Computer generated images are typically made up of many differing components or graphical elements which are rendered and composited together to create a final image. In recent times, an "opacity channel" (also known as a "matte", an "alpha channel", or simply "opacity") has been commonly used. The opacity channel contains information regarding the transparent nature of each element. The opacity channel is stored alongside each instance of a colour, so that, for example, a pixel-based image with opacity stores an opacity value as part of the representation of each pixel. An element without explicit opacity channel information is typically understood to be fully opaque within some defined bounds of the element, and assumed to be completely transparent outside those bounds.

An expression tree offers a systematic means for representating an image in terms of its constituent elements and which facilitates later rendering. Expression trees typically comprise a plurality of nodes including leaf nodes, unary nodes and binary nodes. Nodes of higher degree, or of alternative definition may also be used. A leaf node, being the outer most node of an expression tree, has no descendent nodes and represents a primitive constituent of an image. Unary nodes represent an operation which modifies the pixel data coming out of the part of the tree below the unary operator. Unary nodes include such operations as colour conversions, convolutions (blurring etc) and operations such as red-eye removal. A binary node typically branches to left and right subtrees, wherein each subtree is itself an expression tree comprising at least one leaf node. Binary nodes represent an operation which combines the pixel data of its two children to form a single result. For example, a binary node may be one of the standard "compositing operators" such as OVER, IN, OUT, ATOP and alpha-XOR, examples of which and other are seen in Fig. 20.

Several of the above types of nodes may be combined to form a compositing tree. An example of this is shown in Fig. 1. The result of the left-hand side of the compositing tree may be interpreted as a colour converted image being clipped to spline boundaries.

This construct is composited with a second image.

Although the non-transparent area of a graphical element may of itself be of a certain size, it need not be entirely visible in a final image, or only a portion of the element may have an effect on the final image. For example, assume an image of a certain size is to be displayed on a display. If the image is positioned so that only the top left corner of the image is displayed by the display device, the remainder of the image is not displayed. The final image as displayed on the display device thus comprises the visible portion of the image, and the invisible portion in such a case need not be rendered.

Another way in which only a portion of an element may have an effect is when the portion is obscured by another element. For example, a final image to be displayed (or rendered) may comprise one or more opaque graphical elements, some of which obscure other graphical elements. Hence, the obscured elements have no effect on the final image.

A conventional compositing model considers each node to be conceptually infinite in extent. Therefore, to construct the final image, a conventional system would apply a compositing equation at every pixel of the output image. Interactive frame rates of the order greater than 15 frames per second can be achieved by relatively brute-force approaches in most current systems, because the actual pixel operations are quite simple and can be highly optimised. This highly optimised code is fast enough to produce acceptable frame rates without requiring complex code. However, this is certainly not true in a compositing environment.

The per-pixel cost of compositing is quite high. This is because typically an image is rendered in 24-bit colour in addition to an 8-bit alpha channel, thus giving 32 bits per pixel. Each compositing operator has to deal with each of the four channels. Therefore, the approach of completely generating every pixel of every required frame when needed is inefficient, because the per-pixel cost is too high.

Problems arise with prior art methods when rendering graphical objects which include transparent and partially-transparent areas. Further, such methods typically do not handle the full range of compositing operators.

Summary of the Invention

It is an object of the present invention to substantially overcome, or ameliorate, one or more of the deficiencies of the above mentioned methods by the provision of a method for creating an image made up by compositing multiple components.

According to one aspect of the present invention there is provided a method of creating an image, said image to be formed by rendering and compositing at least a plurality of graphical objects, each said object having a predetermined outline, said method comprising the steps of:

dividing a space in which said outlines are defined into a plurality regions, each said region being defined by at least one region outline substantially following at least one of said predetermined outlines or parts thereof and being substantially formed by segments of a virtual grid encompassing said space;

manipulating said regions to determine a plurality of further regions, wherein each said further region has a corresponding compositing expression;

classifying said further regions according to at least one attribute of said graphical objects within said further regions;

modifying each said corresponding compositing expression according to a classification of each said further region to form an augmented compositing expression for each said further region; and

compositing said image using each of said augmented compositing expressions.

According to another aspect of the present invention there is provided a method of method of creating an image, said image to be formed by rendering and compositing at least a plurality of graphical objects, each said object having a predetermined outline, said method comprising the steps of:

dividing a space in which said outlines are defined into a plurality regions, each said region being defined by at least one region outline substantially following at least one of said predetermined outlines or parts thereof and being substantially formed by segments of a virtual grid encompassing said space, wherein each object has two region outlines arranged either side of said predetermined outline to thus define three regions for each said object, and wherein each said region has a corresponding compositing expression;

classifying said regions according to at least one attribute of said graphical objects within said regions;

modifying each said corresponding compositing expression according to a classification of each said region to form an augmented compositing expression for each said region; and

compositing said image using each of said augmented compositing expressions.

According to still another aspect of the present invention there is provided an apparatus for creating an image, said image to be formed by rendering and compositing at least a plurality of graphical objects, each said object having a predetermined outline, said apparatus comprising:

dividing means for dividing a space in which said outlines are defined into a plurality regions, each said region being defined by at least one region outline substantially following at least one of said predetermined outlines or parts thereof and being substantially formed by segments of a virtual grid encompassing said space;

manipulating means for manipulating said regions to determine a plurality of further regions, wherein each said further region has a corresponding compositing expression;

classifying means for classifying said further regions according to at least one attribute of said graphical objects within said further regions;

modifying means for modifying each said corresponding compositing expression according to a classification of each said further region to form an augmented compositing expression for each said further region; and

compositing means for compositing said image using each of said augmented compositing expressions.

According to still another aspect of the present invention there is provided an apparatus for creating an image, said image to be formed by rendering and compositing at least a plurality of graphical objects, each said object having a predetermined outline, said apparatus comprising:

dividing means for dividing a space in which said outlines are defined into a plurality regions, each said region being defined by at least one region outline substantially following at least one of said predetermined outlines or parts thereof and being substantially formed by segments of a virtual grid encompassing said space, wherein each object has two region outlines arranged either side of said predetermined outline to thus define three regions for each said object, and wherein each said region has a corresponding compositing expression;

classifying means for classifying said regions according to at least one attribute of said graphical objects within said regions;

modifying means for modifying each said corresponding compositing expression according to a classification of each said region to form an augmented compositing expression for each said region; and

compositing means for compositing said image using each of said augmented compositing expressions.

According to still another aspect of the present invention there is provided a computer program product including a computer readable medium having a plurality of software modules for creating an image, said image to be formed by rendering and compositing at least a plurality of graphical objects, each said object having a predetermined outline, said computer program product comprising:

dividing module for dividing a space in which said outlines are defined into a plurality regions, each said region being defined by at least one region outline substantially following at least one of said predetermined outlines or parts thereof and being substantially formed by segments of a virtual grid encompassing said space;

manipulating module for manipulating said regions to determine a plurality of further regions, wherein each said further region has a corresponding compositing expression;

classifying module for classifying said further regions according to at least one attribute of said graphical objects within said further regions;

modifying module for modifying each said corresponding compositing expression according to a classification of each said further region to form an augmented compositing expression for each said further region; and

compositing module for compositing said image using each of said augmented compositing expressions.

According to still another aspect of the present invention there is provided a computer program product including a computer readable medium having a plurality of software modules for creating an image, said image to be formed by rendering and compositing at least a plurality of graphical objects, each said object having a predetermined outline, said computer program product comprising:

dividing module for dividing a space in which said outlines are defined into a plurality regions, each said region being defined by at least one region outline substantially following at least one of said predetermined outlines or parts thereof and being substantially formed by segments of a virtual grid encompassing said space, wherein each object has two region outlines arranged either side of said predetermined outline to thus define three regions for each said object, and wherein each said region has a corresponding compositing expression;

classifying module for classifying said regions according to at least one attribute of said graphical objects within said regions;

modifying module for modifying each said corresponding compositing expression according to a classification of each said region to form an augmented compositing expression for each said region; and

compositing module for compositing said image using each of said augmented compositing expressions.

Brief Description of the Drawings

A preferred embodiment of the present invention will now be described with reference to the following drawings:

Fig. 1 is an example of a compositing tree;

Fig. 2 illustrates an image containing a number of overlapping objects and the corresponding compositing tree;

Fig. 3 shows the image of Fig. 2 illustrating the different regions which exist in the image and listing the compositing expression which would be used to generate the pixel data for each region;

Fig. 4 is the image of Fig. 3, illustrating the compositing operations after being optimised according to one example of the preferred embodiment;

Fig. 5 illustrates the result of combining two region descriptions using the Union operation according to the preferred embodiment;

Fig. 6 illustrates the result of combining two region descriptions using the Intersection operation according to the preferred embodiment;

Fig. 7 illustrates the result of combining two region descriptions using the Difference operation according to the preferred embodiment;

Figs. 8A to 8D illustrate the steps involved in combining two region groups using the Over operation according to the present invention;

Fig. 9 illustrates an image and compositing tree according to another example of the preferred embodiment;

Fig. 10 illustrates an image and compositing tree according to still another example of the preferred embodiment;

Fig. 11 illustrates the effect on the image of Fig. 10 of moving region A;

Fig. 12 illustrates an image and compositing tree according to still another example of the preferred embodiment;

Fig. 13 illustrates the effect on the image of Fig. 12 of moving region A;

Fig. 14 illustrates the effect on the image of Fig. 12 of moving region B; and

Fig. 15 illustrates those nodes in a compositing tree which need to have their region groups updated if leaf nodes B and H change;

Fig. 16 illustrates a region and its x and y co-ordinates;

Fig. 17 illustrates two regions and their x and y co-ordinates;

Fig. 18 illustrates an image and compositing tree according to still another example of the preferred embodiment;

Fig. 19 illustrates an apparatus upon which the preferred embodiment is implemented;

Fig. 20 depicts the result of a variety of compositing operators useful with the present invention;

Fig. 21 illustrates regions formed by combining two circles with non-grid-aligned regions;

Fig. 22 illustrates improved regions formed by combining two circles with grid-aligned regions;

Fig. 23 is a flowchart showing a method of creating an image in accordance with the preferred embodiment; and

Appendix 1 is a listing of source code according to the present invention

Detailed Description

1.0 Underlying Principles

The basic shape of operands to compositing operators in most current systems is the rectangle, regardless of the actual shape of the object being composited. It is extremely easy to write an operator which composites within the intersection area of two bounding boxes. However, as a bounding box typically does not accurately represent the actual bounds of a graphical object, this method results in a lot of unnecessary compositing of completely transparent pixels over completely transparent pixels. Furthermore, when the typical make-up of a composition is examined, it can be noticed that areas of many of the objects are completely opaque. This opaqueness can be exploited during the compositing operation. However, these areas of complete opaqueness are usually non-rectangular and so are difficult to exploit using compositing arguments described by bounding boxes. If irregular regions are used for exploiting opaque objects when compositing, then these regions could then be combined in some way to determine where compositing should occur. Furthermore, if any such region is known to be fully transparent or fully opaque, further optimisations are possible.

Most current systems fail to exploit similarities in composition between one frame and the next. It is rare for everything to change from frame to frame and therefore large areas of a compositing tree will remain unchanged. An example of this is where a cartoon type character comprising multiple graphical objects is rendered on a display. If, for example, the character spilt some paint on its shirt in the next frame, then it is not necessary to render the entire image again. For example, the head and legs of the character may remain the same. It is only necessary to render those components of the image that have been altered by the action. In this instance, the part of the shirt on which the paint has been spilt may be re-rendered to be the same colour as the paint, whilst the remainder of the character stays the same. Exploiting this principle may provide large efficiency improvements. If incremental changes are made to the compositing tree, then only a reduced amount of updating is necessary to affect the change.

Many current graphical systems use what is known as an *immediate mode* application program interface (API). This means that for each frame to be rendered, the complete set of rendering commands is sent to the API. However, sending the complete set of rendering commands is somewhat inefficient in a compositing environment, as typically, large sections of the compositing tree will be unchanged from one frame to the next, but would be completely re-rendered anyway in immediate mode. The preferred embodiment, on the other hand, is considered by the present inventors to be best described as a *retained mode* API. Retained mode means that instead of providing the complete compositing tree on a per-frame basis, the user provides an initial compositing tree, and then modifies it on a per-frame basis to effect change. Changes which can be made to the tree include geometrically transforming part or all of the tree, modifying the tree structure (unlinking and linking subtrees), and modifying attributes (eg: color) of individual nodes. Note that such modifications may not necessarily mean that the tree structure, for example as seen in Fig. 1, will change where only the attributes of an individual node have been modified.

The rendering operation of the preferred embodiment is a combination of a number of techniques and assumptions which combine to provide high quality images and high frame rates. Some of the contributing principles are:

(i) The use of irregular regions to minimise per-pixel compositing. For example, if one graphical object is on top of another, then pixel compositing is only needed inside the area where the two objects intersect. Having the ability to use irregular regions gives the ability to narrow down areas of interest much more accurately.

(ii) An assumption is made that in the transition from one frame to the next, only part of the tree will change. This can be exploited by caching away expensive-to-generate information regarding the composition so that it can be re-used from one frame to the next. Examples of expensive-to-generate information are - regions of interest (boundaries of areas of intersection between objects etc); pixel data (representing expensive composites etc); and topological relationships between objects.

(iii) If an opaque object is composited with another object using the OVER operator, then the opaque object completely obscures what it is composited onto (inside the opaque objects area). This is a very useful property because it means that no expensive pixel compositing is required to achieve the output pixel within the area of overlap. (The pixel value is the same as that at the equivalent spot on the opaque object). Opaque objects induce similar behaviour in most of the compositing operators. Therefore, the preferred embodiment attempts to exploit opaque areas as much as possible.

Fig. 23 is a flowchart showing a method of creating an image in accordance with the preferred embodiment of the present invention. The image is formed by rendering graphical objects whereby each of the objects has a predetermined boundary outline. The process begins at step 2301, where a space in which the object outlines are defined is divided into a number of regions. Each of the regions is defined by at least one of the predetermined boundary outlines or parts thereof. The regions are formed by segments of a grid which encompasses the space in which the predetermined outlines are defined. At the next step 2303, the regions are manipulated to determine a number of further regions. Each of the further regions has a corresponding compositing expression. The process of dividing the space into a number of regions and manipulating those regions is described in detail particularly with reference to section 2.3 below. Section 2.3 includes two pseudocode listings which describe steps 2301 and 2303 for the "OVER" and "IN" compositing operations. The process continues at step 2305, where the further regions are classified according to attributes of the objects that fall within the further regions. At the next step 2307, each of the corresponding compositing expressions are modified according to a classification of each of the further regions. The modifications form an augmented compositing expression for each of the further regions. The process of classifying the further regions and modifying each of the corresponding compositing expressions is described in detail particularly with reference to section 2.4 below. Section 2.4 includes two pseudocode listings which describe steps 2305 and 2207 for the "OVER" and "IN" compositing operations. The process concludes at step 2309, where the image is composited using each of the augmented compositing expressions. Step 2309 is described in detail with reference to section 2.6, below, which includes a pseudocode listing demonstrating the compositing process.

2.0 Basic Static Rendering

Static Rendering deals with the problem of generating a single image from a compositing tree as quickly as possible. Some of the pixel compositing methods of the preferred embodiment will be explained using a static rendering example.

An example of a simple compositing tree which consists of leaf node objects and only using the "OVER" operator is shown in Fig. 2. Conventionally, each node is considered to be conceptually infinite in extent. One method to construct the final image is to apply the compositing equation (((D OVER B) OVER C) OVER (A OVER E)) at every pixel of the output image. However, this is quite an inefficient method.

A composition can generally be subdivided into a number of mutually exclusive irregular regions. The above compositing expression may be simplified independently within each region. In the example of Fig. 2, A, C and E represent opaque objects. B and D, on the other hand are partially transparent. Fig. 3 shows the different regions (1-10) produced using the five objects which exist in the example, and the compositing expression which would be used to generate the pixel data for each specific region.

The compositing expressions provided in Fig. 3 make no attempt to exploit the properties of the object's opacity. If these properties are used to simplify the compositing expressions for each region, the expressions of Fig. 4 are obtained resulting in a simplification of the rendering of regions 2, 3, 5, 6, 7, 8 and 9 compared with Fig. 3. These simplified compositing expressions would result in far fewer pixel compositing operations being performed to produce the final picture.

Fig. 4 represents the region subdivision for the root of the compositing tree. However, every node in the compositing tree can itself be considered the root of a complete compositing tree. Therefore, every node in the compositing tree can have associated with it a group of regions which together represent the region subdivision of the subtree of which the node is the root. Region subdivision provides a convenient means of managing the complexity of a compositing tree and an efficient framework for caching expensive data.

Using the principles noted above, a compositing expression can be simplified dependent upon whether the graphical objects being composited are wholly opaque, wholly transparent or otherwise (herewith deemed "ordinary").

Table 1 shows how the compositing operations of Fig. 20 can be simplified when one or both operands are opaque or transparent.

TABLE 1

Expression	A's opacity	B's opacity	Optimised
AoverB	Transparent	Transparent	neither
	Transparent	Ordinary	B
	Transparent	Opaque	B
	Ordinary	Transparent	A
	Ordinary	Ordinary	AoverB
	Ordinary	Opaque	AoverB
	Opaque	Transparent	A
	Opaque	Ordinary	A
	Opaque	Opaque	A

AroverB	Transparent	Transparent	neither
	Transparent	Ordinary	B
	Transparent	Opaque	B
	Ordinary	Transparent	A
	Ordinary	Ordinary	BoverA
	Ordinary	Opaque	B
	Opaque	Transparent	A
	Opaque	Ordinary	BoverA
	Opaque	Opaque	B
AinB	Transparent	Transparent	neither
	Transparent	Ordinary	neither
	Transparent	Opaque	neither
	Ordinary	Transparent	neither
	Ordinary	Ordinary	AinB
	Ordinary	Opaque	A
	Opaque	Transparent	neither
	Opaque	Ordinary	AinB
	Opaque	Opaque	A
ArinB	Transparent	Transparent	neither
	Transparent	Ordinary	neither
	Transparent	Opaque	neither
	Ordinary	Transparent	neither
	Ordinary	Ordinary	BinA
	Ordinary	Opaque	BinA
	Opaque	Transparent	neither
	Opaque	Ordinary	B
	Opaque	Opaque	B
AoutB	Transparent	Transparent	neither
	Transparent	Ordinary	neither
	Transparent	Opaque	neither
	Ordinary	Transparent	A
	Ordinary	Ordinary	AoutB
	Ordinary	Opaque	neither
	Opaque	Transparent	A
	Opaque	Ordinary	AoutB
	Opaque	Opaque	neither
AroutB	Transparent	Transparent	neither
	Transparent	Ordinary	B
	Transparent	Opaque	B
	Ordinary	Transparent	neither
	Ordinary	Ordinary	BoutA
	Ordinary	Opaque	BoutA
	Opaque	Transparent	neither
	Opaque	Ordinary	neither
	Opaque	Opaque	neither
AatopB	Transparent	Transparent	neither
	Transparent	Ordinary	B
	Transparent	Opaque	B

	Ordinary	Transparent	neither
	Ordinary	Ordinary	AatopB
	Ordinary	Opaque	AatopB
	Opaque	Transparent	neither
	Opaque	Ordinary	AatopB
	Opaque	Opaque	A
AratopB	Transparent	Transparent	neither
	Transparent	Ordinary	neither
	Transparent	Opaque	neither
	Ordinary	Transparent	A
	Ordinary	Ordinary	BatopA
	Ordinary	Opaque	BatopA
	Opaque	Transparent	A
	Opaque	Ordinary	BatopA
	Opaque	Opaque	B
AxorB	Transparent	Transparent	neither
	Transparent	Ordinary	B
	Transparent	Opaque	B
	Ordinary	Transparent	A
	Ordinary	Ordinary	AxorB
	Ordinary	Opaque	AxorB
	Opaque	Transparent	A
	Opaque	Ordinary	AxorB
	Opaque	Opaque	neither

2.1 Basic Data Model

Associated with every node in a compositing tree is a group of mutually exclusive regions which together represent the non-transparent area of the node. It should be noted that the region descriptions that the preferred embodiment uses are generally not pixel accurate. A region may in fact contain some transparent pixels. However, any point lying outside of all the regions at a node is certain to be transparent. The set of the mutually exclusive regions at a node is known as a *region group*. A leaf node region group may contain only one or two regions. The region group at the root of the tree may contain hundreds of regions. Each region in a region group contains the following basic data:

(i) **A Region Description** is a low-level representation of the boundaries of the region. The region descriptions of all the regions in a region group must be mutually exclusive (non-intersecting). However, the preferred embodiment is not limited to using axis-parallel (ie: every side parallel or perpendicular to a scan line of an output device) region descriptions. The preferred embodiment allows region descriptions which more closely represent arbitrary shaped regions.

(ii) A Proxy is some means of caching the pixel data resulting from applying the operations specified by the compositing expression at every pixel inside the region description. A proxy can be as simple as a 24-bit colour bitmap, or something much more complicated (such as a run-length encoded description). Fundamentally, a proxy simply has to represent pixel data in some way which makes it efficient to retrieve and use.

Every region group also contains a region description which is the union of all the region descriptions of the regions in the region group. The region description essentially represents the entire coverage of the region group.

2.2 Region Descriptions and Region Arithmetic

The region arithmetic and data structure of the preferred embodiment has the following properties:

-to allow the representation of *complex* regions, including convex regions, concave regions and regions with holes. This is necessary so that a region will be reasonably able to follow the geometry of the graphic object it represents;

-is space efficient. In a complicated composition there will be many regions. For memory efficiency, it is therefore preferable that the cost of storing these regions is reasonably small;

-the region arithmetic should support basic set operations - Union, Intersection and Difference;

-the above-noted basic operations should be efficient in terms of speed. In a complex compositing tree, it is possible that a large amount of region arithmetic will be undertaken. A poor implementation of region arithmetic could lead to the time taken by region arithmetic being greater than the time saved from the reduction in per-pixel compositing;

-it is advantageious if the region description can be geometrically translated efficiently. In cases where a graphic object is translated, the graphics objects associated regions can then be translated quickly; and

-it is sometimes helpful to be able to quickly compare two regions to determine if they are the same. It is not necessary to obtain any other statistics on their similarity, simple equality is all that is required.

Two conventional region description techniques were considered and rejected for the preferred embodiment. These were-

Polygons: A polygon can be used to represent almost any object, the disadvantage of using a polygon, however, is that a ploygon's generality makes implementing the set operations slow and inefficient.

Quadtrees: Using quadtrees, set operations are easy to implement and are quite efficient. In addition, they can represent a wide variety of regions given sufficient granularity (all edges in a quadtree have to be axis-parallel). Their major failing is that all quadtrees must be aligned on the same grid (granularity). This means that it is impossible to simply translate a quadtree by an arbitrary amount. Unless that amount is a multiple of the underlying grid size, the quadtree will need to be recalculated from the object it describes (otherwise it will keep growing). Therefore, quadtrees are not suitable in application domains where geometric translation is a frequent operation.

The region description data structure of the preferred embodiment can be understood by imagining that along a vertical line every coordinate has a state which is one of either inside or outside the region. The data structure stores those y co-ordinates at which some change of state between inside and outside occurs. For each such y co-ordinate, the data contains spans of coordinates each of which toggles the state of every vertical line running through the data. Each span of x co-ordinates is called a run. The sequence of runs associated with a y co-ordinate is called a row. For example, the region of Fig. 16 could be described by the following:

row y = 10 : x = 10, x = 100

row y = 100 : x = 10, x =100

Similarly, the regions of Fig. 17 could be described by the following:

row y = 10 : x = 10, x =100

row y = 30 : x = 30, x = 70

row y = 70 : x = 30, x = 70

row y = 100 : x +10, x = 100

The data representing a region is represented by an array of integer values. There are two "special" values -

R_NEXT_IS_Y	A beginning-of-row marker. Indicates that the next integer in the sequence will represent a y coordinate.
R_EOR	Stands for End-of-Region. Indicates that the region description has finished.

All other values represent x or y coordinates. The x coordinates in a row represent runs. The first two co-ordinates represent a run, then the next two represent the next run and so on. Therefore, the x coordinates in a row should always be increasing. Also, there should always be an even number of x-coordinates in a row. The region data stream for Fig. 17 is shown below.

```
R_NEXT_IS_Y 10 10 100
R_NEXT_IS_Y 30 30 70
R_NEXT_IS_Y 70 30 70
R_NEXT_IS_Y 100 10 100
R_EOR
```

The preferred embodiment also contains the bounding box of the region, as this is useful in certain set operations.

As seen in Fig. 6, if two region descriptions are combined using a Union operation, then the resultant region description will describe an area in which either region description is active.

As seen in Fig. 7, if two region descriptions are combined using the Intersection operation, then the resultant region description will describe an area in which both the region descriptions are active.

If two region descriptions are combined using the Difference operation, then the resultant region will describe an area in which *only the first region is active*, as seen in Fig. 8.

2.3 Constructing Region Groups:

2.3.1 Constructing Leaf Node Region Groups

A region group for a leaf node will typically contain one or more regions, which together fully contain the non-transparent area of the graphical object represented by the leaf node. Typically, the non-transparent area is divided into regions where each region has some property that facilitates optimization. For example, the non-transparent area of some graphical object can be divided into two regions, one fully opaque and the other with ordinary opacity. The above mentioned compositing optimizations would apply where the opaque region is composited.

Alternatively, the leaf node could be subdivided based on some other attribute. For example, a leaf node could be divided into two regions, one representing an area of constant colour, the other representing blended colour. Areas of constant colour may be composited more efficiently than areas with more general colour description.

2.3.1.1 Region Formation and Phasing

When creating regions, it is not always beneficial that region boundaries follow graphical object boundaries precisely. What is important is that any property that facilitates optimization is valid at all points within a region said to have that property.

For example, an opaque circle could be covered exactly by one circular region which is classified as opaque, or by two approximate regions, one fully opaque octagonal region inscribed in the circle, and one annular octagonal region of ordinary opacity that includes the remainder of the circle plus some area exterior to the circle.

There is typically a trade-off between how closely region boundaries follow graphical object boundaries and the benefits obtained. If region boundaries follow object boundaries very closely, a lot of work is usually involved in creating the region boundaries and in performing intersections and differences of regions (the reasons for needing to perform such operations are explained in later sections). However, if region boundaries are too approximate, they may either include large areas that are outside the objects' boundaries, resulting in too much unnecessary compositing, or they may fail to include large areas where known properties lead to optimization.

One approach, as illustrated in the appendix, is to limit region boundaries to sequences of horizontal and vertical segments. Using this approach, the typical segment size is chosen so that there is neither too much detail so that the region operations are overburdened, nor too much approximation to result in wasted compositing or insufficient optimization.

One method to improve the efficiency of region operations is to choose as many as is practical of the horizontal and vertical segments of substantially all region boundaries to be in phase. In other words, the horizontal and vertical segments are to be chosen from the horizontal and vertical lines of the same grid. The grid need not be regularly spaced, nor have the same spacing horizontally and vertically, although typically it will.

Choosing the horizontal and vertical segments from the horizontal and vertical lines of the same grid improves the efficiency of region operations by seeking to keep all region boundary detail to the level of detail contained in the underlying grid. Without constraining the majority of region boundary segments to a grid, region operators such as difference and intersection tend to produce a lot more fine detail. For example, in Fig. 21, two circles 901 and 902 are shown with respective regions 903 and 904 that are not grid-aligned. These circles are overlapped yielding difference regions 905 and 907, and intersection region 906. In Fig. 22, the same circles 901 and 902 have regions 913 and 914 that are aligned to grid 910. These circles are overlapped yielding difference regions 915 and 917 and intersection region 916. It can be seen in this example that the grid-aligned regions yield less detailed results at the expense of slightly less efficient region coverage. Regions 905, 906 and 907 together contain a total of sixty segments, while regions 915, 916 and 917 together contain only fifty-two.

2.3.2 Creating Binary Region Groups

The region groups of binary nodes in the compositing tree on the other hand are the result of combining the region groups of their child nodes. It will now be explained how region groups are combined to form new region groups. In this section, for simplicity only “OVER” and “IN” binary nodes will be dealt with. The operations required for binary nodes representing other compositing operators can easily be inferred from combining the “OVER” and “IN” cases in various ways.

For the sake of clarity, the method of the preferred embodiment is initially described without reference to optimization based properties such as opacity.

The following notation will be beneficial when considering binary region group creation:

Notation

RG1	The region group of the binary node's left child
RG2	The region group of the binary node's right child
RG	The region group of the binary node. It is this region group that is being initialised
RG1→urgn	The region description representing the union of all RG1's region descriptions (RG1's coverage region).
RG2→urgn	The region description representing the union of all RG2's region descriptions (RG2's coverage region).
RG→urgn	The union of all RG's region descriptions (to be initialised) (RG's coverage region)
rg1i	The current region in RG1
rg2j	The current region in RG2
rg1i→rgn	rgli's region description
rg2j→rgn	rg2j's region description
rg1i→proxy	rg1i's proxy
rg2j→proxy	rg2j's proxy

2.3.2.1 Constructing "OVER" Region Groups

When constructing "OVER" region groups, only areas where the contributing region groups intersect need to be composited. Areas where one operand does not overlap the other involve no compositing. The method is broken into three iterative steps. First, the coverage region of the region group of the binary node that is being initialised (RG→urgn) is made equal to the union of the coverage regions of the binary nodes left child (RG1→urgn) and the binary node's right child (RG2→urgn). Then, for each region rg_i in RG1, the difference (diff_rgn) between that region and RG2's coverage region (RG2→urgn) is then calculated. If the difference (diff_rgn) is non-empty then a new region with diff_rgn as its region description is added to RG. The proxy of this new difference region can be the same as the proxy $rg1_i$. No compositing is required to generate it. The difference regions between RG2's regions and RG1's coverage region are similarly constructed and added to RG. Finally, the intersection (inter_rgn) between each region $rg1_i$ in RG1 and each region $rg2_j$ in RG2 is calculated. If the result of this intersection is non-empty, then a new proxy (new_p) is created by compositing $rg1_i$'s proxy with $rg2_j$'s proxy using the over operation with the inter_rgn. A new region is then added to RG with inter_rgn as its region description and new_p as its proxy. The method of constructing "OVER" groups in accordance with the preferred embodiment is described below using pseudo-code.

```
RG→urgn = RG1→urgn union RG2→urgn
FOR i = 0 TO number of regions in RG1 DO
    diff_rgn = rg1_i→rgn difference RG2→urgn
    IF diff_rgn is non-empty THEN
        ADD to RG a new region with diff_rgn as its region description and
rg1_i→proxy as its proxy. (*)
    END IF
    FOR j = 0 TO number of regions in RG2 DO
        inter_rgn = rg1_i→rgn intersection rg2_j→rgn
        IF inter_rgn is non-empty THEN
            create new proxy new_p initialised to OVER of rg1_i→proxy and
rg2_j→proxy inside inter_rgn.
            ADD to RG a new region with inter_rgn as its region description
and new_p as its proxy. (+)
        END IF
    END DO
END DO
FOR j = 0 TO number of regions in RG2 DO
    diff_rgn = rg2_j→rgn difference RG1→urgn
    IF diff_rgn is non-empty THEN
        ADD to RG a new region with diff_rgn as its region description and
```

```
rg2$_j$→proxy as its proxy. (*)
        END IF
END DO
```

The regions added by the ADD operations marked with asterisks (*) above are termed difference regions since their shape is the result of a difference operation. Such regions are very cheap computationally because their proxies require no compositing. The only work involved is the administrative overhead of adding a new region to the region group and the cost of the difference operation itself. In accordance with the preferred embodiment, a proxy is inherited from the region (in one of the child region groups) on which it is based. It can be seen that proxies which originate low in the compositing tree can be propagated upwards towards the root with minimal overhead (both in terms of speed and memory) by the use of difference regions.

The regions added by the ADD operation marked with the plus (+) are termed intersection regions. This is because their shape is the result of an intersection operation. The proxies of such regions are more expensive to generate than difference regions because they involve per-pixel compositing operations to be done within the area defined by the intersection. The more fidelity granted the region descriptions, the greater the saving in pixel processing costs, at the cost of a greater administrative overhead (more complex regions require longer to intersect etc).

Figs. 8A to 8D provide a simple example of combining "OVER" region groups using the above method. The region group resulting from the combination contains 5 regions, 3 difference regions and 2 are intersection regions. Fig. 8A represents two region groups RG1 and RG2 which are to be combined. RG1 contains two regions 81 and 82, whereas RG2 only contains a single region 83. As seen in Fig 8B, for each region in RG1, RG2's region coverage is subtracted from it. If the resultant region is non-empty, the resultant region becomes a region in the new region group. In this example both regions 81 and 83 produce non-empty difference regions 84 and 85 respectively. For each region in RG2, RG1's region coverage is subtracted from it, as seen in Fig 8C. In this example difference region 86 is produced. Finally, every region in RG1 is intersected with every region in RG2, as seen in Fig 8D. Any non-empty region becomes a region in the new region group. In this example, regions 81 and 83 produce 87. Further, regions 82 and 83 produce 88.

2.3.2.2 Constructing "IN" Region Groups

The properties of the "IN" operator lead to the fact that an "IN" binary region group only produces pixel data in the region of intersection between the two contributing region groups. Essentially, when compared to the algorithm used for "OVER" region groups, only intersection regions are generated. Therefore, for each region $rg1_i$ of RG1, and for each region $rg2_j$ of RG2 the intersection ($inter\_rgn_{ij}$) between $rg1_i$ and $rg2_j$ is calculated. If the intersection is non-empty then a new proxy (new_p) is created by compositing $rg1_i$'s proxy with $rg2_j$'s proxy using the "in" operation within $inter\_rgn_{ij}$. A new region is then added to RG with inter_rgn as its region description and new_p as its proxy. The pseudocode describing the method of constructing "IN" region groups in accordance to the preferred embodiment is provided below:

```
RG→urgn = RG1→urgn intersection RG2→urgn
FOR i = 0 TO number of regions in RG1 DO
    FOR j = 0 TO number of regions in RG2 DO
        inter_rgn = $rg1_i$→rgn intersection $rg2_j$→rgn
        IF inter_rgn is non-empty THEN
            create new proxy new_p initialised to IN of $rg1_i$→proxy and
$rg2_j$→proxy inside inter_rgn.
            ADD to RG a new region with inter_rgn as its region description
and new_p as its proxy. (+)
        END IF
    END DO
END DO
```

The major difference between the "IN" and the "OVER" cases is that the "OVER" case generates difference regions while "IN" does not. In the example demonstrated by Figs. 8A to 8D, only new regions 97 and 98 would be generated, as these are intersection regions. Difference regions 94, 95 and 96 would not be generated using "IN".

Using **Table 2** below and the pseudocode examples of "OVER" and "IN", the relevant code for other compositing operators can be derived.

2.3.2.3 Constructing Region Groups of Other Compositing Operators

Other compositing operators typically generate the same intersection regions as the "OVER" and "IN" cases do. However, they typically differ from one another (as indeed from "OVER" and "IN") in what difference regions they generate. This is dependent on the particular properties of each compositing operator. Table 2 summarises which difference regions are generated for some commonly used compositing operators.

TABLE 2

Compositing Operator	Generate Diff Rgns from RG1 ?	Generate Diff Rgns from RG2 ?
Over	Yes	Yes
In	No	No
Out	Yes	No
Atop	No	Yes
Xor	Yes	Yes
Plus	Yes	Yes

2.4 Optimising using Opaque Areas

The preferred embodiment stores within each region a flag indicating whether the pixel data in the region proxy is completely opaque. It is therefore possible to reduce the number of per-pixel compositing operations by exploiting the effect opaque operands have on the compositing operators.

2.4.1 Opaque Area Optimisation for "Over" Region Groups

If an opaque region is “OVER” another region, then there is no need to compute the result of the composite, as no part of the right operand region’s proxy is visible through the left operand’s opaque proxy. In the preferred embodiment, the resultant region is made to reference the right operand’s proxy, which has the same effect as actually doing the composite.

The method for opaque area optimisation for “OVER” region groups is a slightly modified version of the "OVER" region group construction method provided previously. The only difference is that when calculating the intersection region of the current region in RG1 and each region of RG2, a check is carried out to see whether the current region in RG1 is opaque. If this is the case, then the proxy of the newly calculated region (new_p) will be the proxy of the current region in RG1.

The method is illustrated using the following pseudocode :

```
RG→urgn = RG1→urgn union RG2→urgn
FOR i = 0 TO number of regions in RG1 DO
    diff_rgn = rg1_i→rgn difference RG2→urgn
    IF diff_rgn is non-empty THEN
        ADD to RG a new region with diff_rgn as its region description and
rg1_i→proxy as its proxy. (*)
    END IF
    FOR j = 0 TO number of regions in RG2 DO
        inter_rgn = $rg1_i$→rgn intersection $rg2_j$→rgn
        IF inter_rgn is non-empty THEN
            IF $rg1_i$ is OPAQUE THEN
                new_p = $rg1_i$→proxy
            ELSE
                create new proxy new_p initialised to OVER of $rg1_i$→proxy and $rg2_j$→proxy inside inter_rgn.
            END IF
            ADD to RG a new region with inter_rgn as its region description and new_p as its proxy. (+)
        END IF
    END DO
END DO
FOR j = 0 TO number of regions in RG2 DO
    diff_rgn = $rg2_j$→rgn difference RG1→urgn
    IF diff_rgn is non-empty THEN
        ADD to RG a new region with diff_rgn as its region description and $rg2_j$→proxy as its proxy. (*)
    END IF
END DO
```

2.4.2 Opaque Area Optimisation for "IN" Region Groups

If a region is "IN" an opaque region, then according to the properties of the "IN" operator, the resultant pixel data is the same as that of the left operand. This can be achieved by having the resultant region simply reference the proxy of the left operand. The method of the preferred embodiment is a slightly modified version of the "IN" region group construction method provided previously. The only difference is that when calculating the intersection region of the current region in RG1 and each region of RG2, a check is carried out to see whether the current region in RG2 is opaque. If this is the case, then the proxy of the newly calculated region (new_p) will be the proxy of the current region in RG1.

The technique is illustrated using the following pseudocode:

```
RG→urgn = RG1→urgn intersection RG2→urgn
FOR i = 0 TO number of regions in RG1 DO
    FOR j = 0 TO number of regions in RG2 DO
        inter_rgn = rg1_i→rgn intersection rg2_j→rgn
        IF inter_rgn is non-empty THEN
            IF rg2_j is OPAQUE THEN
                new_p = rg1_i→proxy
            ELSE
                create new proxy new_p initialised to IN of rg1_i→proxy and
rg2_j→proxy inside inter_rgn.
            END IF
            ADD to RG a new region with inter_rgn as its region description
and new_p as its proxy. (+)
        END IF
    END DO
END DO
```

2.5 Initialising the Entire Tree

The entire compositing tree can be initialised by using the above-described method of the preferred embodiment on every binary region group in the tree. A node cannot be initialised until its children have been initialised. Therefore the process simply starts at the bottom of the tree and works its way up towards the root. The process first checks to see if the current node is a leaf node. If this is the case, then a leaf node region group is constructed. However, in the case that the current node is a binary node then a binary node region group is constructed using the method of the preferred embodiment outlined in sections 2.4.1 and 2.4.2. The following pseudocode outlines a method for initialising all the region groups of the tree. The method utilises a recursive function, which is called passing the root of the tree as an argument.

```
tree_init(node : tree ptr)
BEGIN
    IFnode is a leaf node THEN
        CONSTRUCT leaf node region group
    ELSE
        tree_init(node→left)
```

```
            tree_init(node→right)
            CONSTRUCT binary node region group by combining region groups of the left and right children
      END IF
END tree_init
```

2.6 Constructing the Resultant Image

Once the compositing tree has been initialised, the region group at the root of the tree contains a group of zero or more regions which together represent the partitioning of the resultant image into areas which differ in the way the image data was generated. Some of the regions' proxies can refer to image data directly from leaf nodes of the tree, having not required any compositing. Other regions, on the other hand, may have proxies which are the result of compositing operations. If a single resultant image is required, such as an image stored in a pixel buffer, this can be achieved by copying the image data from each region's proxy to the pixel buffer within the area corresponding to the region. The process is demonstrated in the pseudocode provided below, which is generalised and able to restrict the construction of the final image to any nominated update region.

```
construct_image
(
      output_image : pixel data ptr,
      urgn : region description
)
BEGIN
      FOR i = 0 TO number of region in RG DO
            int_rgn = $rg_i$→rgn intersection urgn
            IF int_rgn is non-empty THEN
                  COPY image data from $rg_i$→proxy to output_image inside int_rgn
            END IF
      END DO
END construct_image
```

3.0 Dynamic Rendering

Dynamic Rendering refers to the problem of generating multiple successive images. Given a compositing tree, it is possible to generate it's region groups (containing regions and proxies) using the method described above. A further embodiment of the above mentioned preferred method, which supports dynamic rendering is described below. The compositing tree represents an image. Changes to the tree can be made to make the tree represent a new image. The tree's region groups (and tree region description and proxies) are updated to reflect this modified tree. Performance is improved by exploiting commonality between the two images. An example will illustrate the techniques and terminology of the further embodiment.

Fig. 3 shows the region subdivision and the respective compositing expressions (advantage is not taken of opacity) for the simple compositing tree. Consider therefore the situation in which object A moves by a small amount relative to the other objects. Some regions in the region group at the root of the tree will be affected by A moving.

If opaque case optimisations are ignored, the regions with compositing expressions which include A will be significantly affected by A moving. The region numbers which are so affected are 2, 3, 5 and 6. When updating the region group at the root of the tree, those regions will need both their region descriptions and their proxies completely recalculated. This situation is known in the further embodiment as primary damage. Any region whose compositing equation includes an object which has changed in some way, may be said to suffer primary damage.

Regions that abut regions which have A in their compositing expression are also effected by A moving, though not as severely as those regions with primary damage. In the example, these other affected regions are 1, 4, 7 and 8. When updating the region group at the root of the tree, these regions will need their region descriptions recalculated. However, their proxies will only need to be recalculated in areas of the new region which were not included in the corresponding earlier region. This situation is known in the further embodiment as secondary damage. Generally, secondary damage is incurred if an object upon which a region's boundary (but not content) depends, changes in some way.

In order to reduce the per-frame update cost, it is important to reduce, as far as is practicable, the amount of work necessary, both in terms of per-pixel operations, but also in terms of region group operations. The concepts of primary and secondary damage are a way of facilitating this. If the preferred embodiment is able to accurately determine the minimum set of regions throughout all the compositing tree which have some kind of damage, then obviously the amount of work being done is reduced. The following sections describe how the reduction in work done is achieved.

3.1 Basic Data Model

The data model used for static rendering, consisting as it does of a region description and a proxy, is insufficient for use in dynamic rendering. This is because, for primary and secondary damage to be determined, it must be possible to associate regions of the same content between frames. To support the association of regions of the same content, some extra information is required in each region in a region group. Therefore, each region in a region group now contains the following data:

(i) A Region Description: A low-level representation of the boundaries of the region. The region descriptions of all the regions in a region group must be mutually exclusive (non-intersecting, non-overlapping).

(ii) A Proxy: Some means of caching the pixel data resulting from applying the operation specified by the compositing expression at every pixel inside the region description. A proxy can be as simple as a 24-bit colour bit-map, or something much more complicated (such as a run-length encoded description). Fundamentally, a proxy simply has to represent pixel data in some way which makes it efficient to retrieve and use.

iii) A Contents Label: A contents label represents a unique symbolic expression that describes the method of construction of image data. The terms in the symbolic expression distinguish between different categorisations of a source of image data. Therefore, the region groups of two distinct leaf nodes in the compositing tree will contain regions which are labelled with distinct contents labels even if their actual image data is equivalent. The further embodiment uses a system of unique integers to represent --------contents labels. For example "23" could represent "(A over B) over C".

(iv) A Flag Register: A general-purpose flag register used to store state during the region group update process. The exact flags stored here will be outlined in a later section.

3.2 Contents Labels

Leaf node region groups can contain multiple regions, with each region naturally having a unique contents label. For example, the region group of a leaf node in a compositing tree could contain a single region (tagged with a single contents label) representing the non-transparent area of the leaf node. Alternatively, the leaf node region group could contain two regions (each tagged with a different contents label), one representing the area of the leaf node which is completely opaque, the other representing the remaining non-transparent area. A leaf node can also be categorised even further, into an arbitrary number of regions (and associated contents labels).

One way a contents label can be created is by assigning a new one to a region of a leaf node region group. Another way is taking other contents labels and combining them to create a new contents label that represents the symbolic expression that represents the combination of the contributing expressions. For example, if the contents label representing ((A comp B) comp C) is combined with the contents label representing (D comp E) then a new contents label will be created which represents (((A comp B) comp C) comp (D comp E)).

As well as contents labels, dependency information is also required. Dependency information indicates how a given contents label is related to other contents labels, both in terms of how the contents of one region contribute to contents of other regions, and how change of a region boundary affect the boundary of other regions. The further embodiment associates the following data with each contents label.

(i) Primary Dependency List: Each primary dependency is a contents label L' to which a contents label L directly contributes. In other words, a "primary dependency" is a contents label L' representing an expression which has been constructed by combining L and some other contents label. Each time contents labels are combined, the contents label for the combination is added to the primary dependencies of all contributors.

(ii) Secondary Dependency List: Each secondary dependency is a contents label L" which can be indirectly affected if the image represented by the contents label L has changed in some way that affects it's boundary. Whenever contents labels are combined, a contributing contents label is added to the secondary steps of the continuation if and only if the compositing operator yields a difference region with said contributing contents label. Table 2 shows which of some commonly used operators yield difference regions for their left and right operands. In addition, for a combination of (A comp B), the secondary dependencies of the combination contents labels for all (A comp b_i) and all (a_j comp B) are added, where a_j are the secondary dependencies of A and b_i are the secondary dependencies of B.

(iii) Property Information: Each contents label can represent contents which have properties which the compositing engine may be able to exploit. An example is that of opaqueness. If a contents label represents opaque content, then compositing that content could be much faster, as for certain operators, no per-pixel compositing operations would be required.

3.3 Contents Label Implementation

The further embodiment uses unique integers as contents labels, and stores a number representing the number of contents labels which currently exist. When a new contents label is created, the number is incremented and becomes the unique integer representing the contents label. This technique of assigning a contents label by monotonically incrementing an integer means that the contents labels' associated data structures can be stored in a one dimensional array which grows as more contents labels are added. A content label's data structure can be referenced simply by using the contents label as an index. When a leaf node contents label is created, the contents label is initialised to have no primary or secondary dependencies. If the current leaf node contents label is opaque, then a flag is set in content label i's properties.

The pseudocode below illustrates the basic techniques used to create a new contents label which is not dependent on other contents labels (ie: a leaf node region group contents label):

Notation

opaque	A flag passed to the function which indicates whether or not the contents label represents opaque content or not.
cur_clab	A global integer which stores the last contents label created.
clabs	A global array which stores the associated data structures of the contents label.
clabs[i]->pri_deps	A pointer to the head of content label i's primary dependency list.
clabs[i]->sec_deps	A pointer to the head of content label i's secondary dependency list.
clabs[i]->properties	A flag register representing contents label i's properties.

```
create_new_contents_label
(
    opaque : boolean
) : RETURNS unsigned int
BEGIN
    INCREMENT cur_clab.
    clabs[cur_clab]→pri_deps = NULL.
    clabs[cur_clab]→sec_deps = NULL.
    IF opaque THEN
        clabs[cur_clab]→properties = OPAQUE.
    ELSE
        clabs[cur_clab]→properties = 0.
    END IF
    RETURN cur_clab.
```

```
END create_new_contents_label.
```

Contents labels can also be created to represent the combination of existing contents labels. This is achieved in the further embodiment by a hash table which maps an operation and the contents labels of its operands (hashed together to create a key) to a single contents label representing the result.

When a region is created which represents an intersection between two other regions (each with its own contents label), a new contents label is generated which is used to tag the new region. When this new contents label is generated, it must be added to the primary dependency lists of both its contributing operands. A secondary dependency list which depends on the secondary dependencies of the two contributing contents labels as well as the properties of the compositing operator must also be generated.

The process is recursive and begins by adding the newly created contents label (new_cl) to the primary dependency lists of the contributing contents labels. Then, depending on the properties of the compositing operator, none, either or both of the contributing contents labels are added to the secondary dependency list. Then every contents label representing (clab1 op $sd2_i$) and ($sd1_i$ op tab2) are added to the secondary dependency list.

Notation

clab1	The first contributing contents label.
clab2	The second contributing contents label.
sd1i	The i'th element of clab1's secondary dependency list.
sd2i	The i'th element of clab2's secondary dependency list.

```
create_binary_contents_label
(
    clab1 : contents label,
    clab2 : contents label,
    op: compositing operator
)
BEGIN
    IF the hash table already contains an entry representing clab1 op clab2
THEN
        RETURN the existing contents label representing the combination.
```

```
END IF
Generate a new entry in the hash table representing clab1 op clab2, mapping to new_cl.

(Add the new contents label to the primary dependency lists of the contributing contents labels if the compositing op requires it)
add_to_primary_dep_list(clab1, new_cl)
add_to_primary_dep_list(clab2, new_cl)

(Generate the secondary dependencies)
IF op generates left diff rgns THEN
    add clab1 to secondary deps
END IF
IF op generates right diff rgns THEN
    add clab2 to secondary deps
END IF
FOR i = 0 TO number of elements in sd1 DO
    add_to_secondary_dep_list
    (
        new_cl,
        create_binary_contents_label($sd1_i$, clab2)
    )
END DO

FOR i = 0 TO number of elements in sd2 DO
    add_to_secondary_dep_list
    (
        new_cl,
        create_binary_contents_label(clab1, $sd2_j$)
    )
END DO
END constuct_binary_contents_label
```

3.4 Combining Region Groups for Dynamic Rendering

Before any incremental updates can be made to a compositing tree, the compositing tree must be constructed to be in a consistent initial state. The basic technique for achieving this is the same as that used for static rendering, except that support for contents labels is included.

Leaf node region groups are initialised essentially as with the static rendering case, except that each region in each leaf node region group is tagged with a unique contents label. Each contents label can in turn be tagged with various categorisation properties which may help the renderer to be more efficient. For example, a contents label can be tagged as being completely opaque.

The initialisation of binary nodes is also similar to the static rendering case. By way of example, the way in which the region group for an "OVER" binary node is constructed will now be explained. The techniques for constructing the region groups of the other compositing operators can easily be inferred from the "OVER" case.

When a difference region between rg_i of one operand and the coverage region of the other operand is calculated, the difference region inherits the contents label rg_i. When an intersection region is created, on the other hand, a new contents label is created by combining the contents labels of the two contributing regions since the two contributing regions had their proxies composited into a new proxy which means new content. The pseudocode for constructing an "OVER" region group which includes contents label management is provided below:

Notation

RG1	The region group of the binary node's left child
RG2	The region group of the binary node's right child
RG	The region group of the binary node. It is this region group that we are initialising
RG1→urgn	The region description representing the union of all RG1's region descriptions (RG1's coverage region).
RG1→urgn	The region description representing the union of all RG2's region descriptions (RG2's coverage region).
RG→urgn	The union of all RG's region descriptions.
rg1i	The current region in RG1
rg2j	The current region in RG2
rg1i→rgn	rg1i's region description
rg2j→rgn	rg2j's region description
rg1i→proxy	rg1i's proxy

rg2j→proxy	rg2j's proxy

```
RG→urgn = RG1→urgn union RG2→urgn
FOR i = 0 TO number of regions in RG1 DO
    diff_rgn = $rg1_i$→rgn difference RG2→urgn
    IF diff_rgn is non-empty THEN
        ADD to RG a new region with diff_rgn as its region description, $rg1_i$→proxy as its proxy and $rg1_i$→clab as its contents label.
    END IF
    FOR j = 0 TO number of regions in RG2 DO
        inter_rgn = $rg1_i$→rgn intersection $rg2_j$→rgn
        IF inter_rgn is non-empty THEN
            new_clab = GENERATE a new unique contents label as a result of combining $rg1_i$→clab and $rg2_j$→clab.
            IF $rg1_i$→clab is OPAQUE THEN
                new_p = $rg1_i$→proxy
            ELSE
                create new proxy new_p initialised to OVER of $rg1_i$→proxy and $rg2_j$→proxy inside inter_rgn.
            END IF
            ADD to RG a new region with inter_rgn as its region description, new_p as its proxy and new_clab as its contents label.
        END IF
    END DO
END DO
FOR j = 0 TO number of regions in RG2 DO
    diff_rgn = $rg2_j$→rgn difference RG1→urgn
    IF diff_rgn is non-empty THEN
        ADD to RG a new region with diff_rgn as its region description, $rg2_j$→proxy as its proxy and $rg2_j$→clab as its contents label.
    END IF
END DO
```

3.5 Secondary Dependencies and Over

The rationale behind the preferred method used for generating secondary dependencies requires more explanation. Secondary dependencies are only generated when a new contents label is created by combining two other contents labels. As can be seen in the above pseudocode, this only occurs when an intersection region is generated. Essentially, the further embodiment uses contents labels generated for intersection regions as triggers - the regions tagged with two contents labels cannot indirectly affect one another unless they intersect. The secondary dependency list for a particular contents label is dependent on the compositing operator the composite contents label represents, the two contributing contents labels and their secondary dependency lists.

The method of the further embodiment of generating a secondary dependency list for a new contents label (C) which represents one contents label (A) composited over another contents label (B) using the "OVER" operator will now be explained. Elements of A's and B's secondary dependency lists are referred to as A_i and B_i respectively. First, both A and B are added to C's secondary dependency list. This is because if the region tagged with C changes its boundary, then it is likely that any regions tagged with A and B will need to be recalculated (because their regions are likely to abut C's region). Next, for each element of B's secondary dependency list, each contents labels representing (A OVER B_i) is added. A mapping representing A OVER B_i may not currently exist in the system and needs to be created. A secondary dependency list can contain contents labels which are not represented by any region in a region group. They could come into existance by changes in region boundaries. The rationale is that A intersects B, and therefore it is likely that A also intersects regions tagged with contents labels which exist in B's secondary dependency list. Similarly, for each element of A's secondary dependency list, each contents label representing (A_i OVER B) is added.

3.6 Contents Labels and Damage

The concepts of primary and secondary damage were introduced with reference to Fig. 3 to demonstrate that it is not always necessary to regenerate an entire image as a result of a change to the compositing tree. By keeping track of dependencies between regions of different content, it only becomes necessary to regenerate image data in regions whose contents have become damaged. The following explanation outlines the dependencies and damage for simple compositing tree changes. "Simple" means that only leaf nodes are modified. More complex change scenarios such as tree structure changes etc will be outlined in later sections.

If a leaf node is modified, the contents labels of its affected regions are said to be "primary damaged". Primary-damaging a contents label involves recursively primary-damaging all its primary dependencies. Whenever a contents label is primary-damaged, all its secondary dependencies are non-recursively marked with secondary damage. The process begins by flagging the contents label to be damaged. The following pseudocode demonstrates how contents labels can be damaged:

Notation

clab	The contents label to be damaged
pdi	The i'th element of clab's primary dependency list.
sdi	The i'th element of clab's secondary dependency list.

```
damage_contents_label
(
    clab : contents label,
)
BEGIN
    FLAG clab with PRIMARY damage

    FOR i = 0 TO number of elements in sd DO
        FLAG $sd_i$ with SECONDARY damage
    END DO

    FOR i = 0 TO number of elements in pd DO
        damage_contents_label($pd_i$)
    END DO
END damage_contents_label
```

When a tree update occurs, any region with its contents label marked as having primary damage will need to recalculate both its region boundaries and its proxy. Any region with its contents label marked as having secondary damage will need to recalculate its region description but will only need to recalculate its proxy in areas of the new region that were not included in the earlier region.

3.7 Examples of Contents Labels and Dependencies

In order to clarify the concepts of contents labels and damage, some examples of varying complexity will be presented.

3.7.1 Example 1

Fig. 9 will result in the following contents label table after the compositing tree is initially constructed (Note: in the following table contents labels are represented as unique strings not as integers where "over" has been abbreviated to "o". This is simply for readability.):

Contents Label	Primary Deps.	Secondary Deps.
A	AoB	
B	AoB	
AoB		A, B

If A moves, then AoB will have primary damage, resulting in B having secondary damage.

3.7.2 Example 2

Fig. 10 will result in the following contents label table after the compositing tree is initially constructed:

Contents Label	Primary Deps.	Secondary Deps.
A	AoB, AoC	
B	AoB, BoC	
AoB	AoBoC	A, B
C	AoC, BoC, (AoB)oC	
AoC		A, C
BoC		B, C
(AoB)oC		AoB, C, AoC, BoC

In this example, every object intersects every other object, so if something changes, everything will be damaged in some way - everything which is a primary dependency of the changed object will have primary damage, whereas everything else will have secondary damage.

Fig. 11 illustrates the effect of A moving in a subsequent frame. As can be seen, if A is damaged, the regions defined by A, AoB, AoC and (AoB)oC will each have primary damage. The regions defined by B, C and BoC will each have secondary damage.

3.7.3 Example 3

Fig. 12 will result in the following contents label table after the compositing tree is initially constructed:

Contents Label	Primary Deps.	Secondary Deps.
A	AoB, AoC, AoE, Ao(DoE), AoD	
B	AoB, BoC, BoE	
AoB	AoBoE	A, B
D	DoE, AoD, CoD, (AoC)oD	
E	DoE, AoE, (AoB)oE, BoE, CoE, (BoC)oE, (AoC)oE	
DoE	Ao(DoE), (AoC)o(DoE), Co(DoE)	D, E
C	AoC, BoC, Co(DoE), CoE, CoD	
AoC	AoCoE, (AoC)o(DoE), (AoC)oD	A, C
BoC	(BoC)oE	B, C
AoE		A, E
(AoB)oE		AoB, E, AoE, BoE
BoE		B, E
CoE		C, E
(BoC)oE		BoC, E, BoE, CoE
AoD		A,D
CoD		C,D
(AoC)oE		AoC, E, AoE, CoE
Ao(DoE)		A, DoE, AoD, AoE
Co(DoE)		C, DoE, CoD, CoE
(AoC)o(DoE)		AoC, DoE, Ao(DoE), Co(DoE), (AoC)oD, (AoC)oE
(AoC)oD		AoC, D, AoD, CoD

Since A intersects every other object, if A moves, a large amount of the compositing tree will need to be recomputed. Fig. 13 shows that the only part left alone is the area corresponding to BoC and its dependent BoCoE. To summarise:

- •Primary Damage - A, AoB, AoC, AoE, Ao(DoE), (AoB)oE, (AoC)oE, (AoC)o(DoE), AoD, (AoC)oD
- •Secondary Damage - B, C, E, DoE, BoE, CoE, DoE, CoDoE

On the other hand, if B moves, the amount of damage is less than if A moved. This is because B doesn't intersect D. DoE, Ao(DoE), (AoC)o(DoE), Co(DoE) and (AoC)oE (and their ancestors) are not damaged when B moves. This is shown in Fig. 14. The rest of the damage is summarised as:

- •Primary Damage - B, AoB, BoC, BoE, (AoB)oE, (BoC)oE
- •Secondary Damage - A, E, C, AoE, CoE

The examples presented so far are simple, but they are sufficient to demonstrate that the dependencies techniques presented so far will damage those contents labels which are affected when a particular contents label/s is(are) damaged. In a typical complex composite, it is rare for large numbers of objects to intersect a large number of other objects, meaning that large areas of the compositing tree should be untouched during updates using the above technique.

3.8 Example of Secondary Dependencies and Compositing Operators

Consider a modified version of Example 3 above. Fig. 18 will result in the following contents label table after the compositing tree is initially constructed. Note that AaB represents A ATOP B and AiB represents A IN B etc:

Contents Label	Primary Deps	Secondary Deps
A	AaB	
B	AaB, BoC	
AaB		B
C	BoC, Co(DiE)	
BoC		B, C
D	DiE	
E	DiE	
DiE	Co(DiE)	
Co(DiE)		C, DiE

As seen in Fig. 18, the ATOP operator clips A to B's bounds, meaning that intersections between A and any of C, D or E never occur. Similar things occur with the IN operator. This means that the objects in this scene are less tightly coupled. For example, if A is changed, then only B and AaB are immediately damaged. Similarly, if E is damaged, it is only possible for DiE to be damaged.

3.9 Updating Region Groups

The further embodiment uses the contents label and damage framework to reduce the amount of work that has to be done to make a binary region group consistent with its updated operands during an update. The further embodiment does this by only updating those regions in a region group whose contents labels have primary or secondary damage, adding any new region which comes into existence as a result of the changes made to the compositing tree, and deleting any region in the right group whose contact no longer exists.

Each different binary operator has a different updating function which deals with the specific requirement of that operator. The process of updating region groups is a two-pass process. The first pass updates any intersection regions that have been primary damaged and adds any new intersection regions generated due to the damage. Each region of one operand's region group is intersected with each region of the other operand's region group whenever one or both of their corresponding contents labels are primary damaged. If the intersection is non-empty, then the further embodiment determines if a contents label representing the combination exists. If the contents label doesn't exist, one is created and primary damaged. Note that primary damaging a contents label will mark all it's secondary dependencies with secondary damage.

If a region in the region group is currently tagged with the primary damage contents label, the regions boundary and proxy are updated. If no such region exists in this region group, then a new region keyed by this contents label is added to the region group. A new proxy is generated and assigned to this region along with the right description relating from the intersection operation.

A difference between each region group of one operand and the coverage region of the other operand is calculated whenever the regions contents label has primary or secondary damage. If the difference is non-empty and a region tagged with the contents label exists in the region group, then it's region description and proxy reference are updated. If such a region doesn't exist then a region keyed by the contents label is added to the region group. The added region is assigned as a coverage region of the difference result and references the proxy of current region.

Each region of one operand's region group is interacted with each region of the other operand's region group whenever the contents label representing their combination has secondary damage and no primary damage. If the intersection is non-empty, the region group is searched looking for a region keyed by the contents label. If such a region exists its region description is updated and it's proxy is updated as the difference between the new and old regions. If such a region doesn't exist, then a region keyed by the contents label is created. The created region description is assigned the result of the interaction operation and it's proxy generated.

Pseudocode which illustrates a simple algorithm for updating a binary "OVER" region group is provided below.

Notation

RG1	The region group of the binary node's left child
RG2	The region group of the binary node's right child
RG	The region group of the binary node. It is this region group

	that is being initialised.
RG1→urgn	The region description representing the union of all RG1's region descriptions (RG1's coverage region).
RG1→urgn	The region description representing the union of all RG2's region descriptions (RG2's coverage region).
RG→urgn	The union of all RG's region descriptions.
rg1i	The current region in RG1
rg2j	The current region in RG2
rg1i→rgn	rg1i's region description
rg2j→rgn	rg2j's region description
rg1i→proxy	rg1i's proxy
rg2j→proxy	rg2j's proxy
rg1i→clab	rg1i's contents label
rg2j→clab	rg2j's contents label

```
RG→urgn = RG1→urgn union RG2→urgn

(First Pass - this pass is used to deal with primary damage of intersection regions
        and any new intersection regions generated)
FOR i = 0 TO number of regions in RG1 DO
        FOR j = 0 TO number of regions in RG2 DO
                IF rg1_i→clab has PRIMARY damage OR rg2_j→clab has PRIMARY DAMAGE THEN
                        inter_rgn = rg1_i→rgn intersection rg2_j→rgn
                        IF inter_rgn is non-empty THEN
                                comp_clab = SEARCH for an existing contents label which represents (rg1_i→clab comp rg2_j→clab).
                                IF a region tagged with comp_clab already exists in RG THEN
                                        IF rg1_i→clab is OPAQUE THEN
                                                new_p = rg1_i→proxy
                                        ELSE
                                                create new proxy new_p initialised to OVER of rg1_i→proxy and rg2_j→proxy inside inter_rgn.
                END IF
                MODIFY the existing region to have inter_rgn as its region description and new_p as its proxy.
            ELSE
                new_clab = create_binary_contents_label(rg1$_i$→clab, rg2$_j$→clab).
                IF rg1$_i$→clab is OPAQUE THEN
                    new_p = rg1$_i$→proxy
                ELSE
                    create new proxy new_p  initialised to OVER of rg1$_i$→proxy and rg2$_j$→proxy inside inter_rgn.
                END IF
                damage_contents_label(new_clab)
                ADD to RG a new region with inter_rgn as its region description, new_p as its proxy and new_clab as its contents label. (+)
            END IF
            FLAG the region as being 'RETAIN AFTER UPDATE'
        END IF
    END IF
  END DO
END DO
```

(Second Pass - this pass is used to deal with primary and secondary damage of difference regions and secondary damage of intersection regions)

```
FOR i = 0 TO number of regions in RG1 DO
    IF rg1$_i$→clab has PRIMARY or SECONDARY damage THEN
        diff_rgn = rg1$_i$→rgn difference RG2→urgn
        IF diff_rgn is non-empty THEN
            IF a region tagged with rg1$_i$→clab already exists in RG THEN
                MODIFY it to have diff_rgn as its region description and rg1$_i$→proxy as its proxy.
            ELSE
                ADD to RG a new region with diff_rgn as its region description, rg1$_i$→proxy as its proxy and rg1$_i$→clab as its contents label. (*)
            END IF
```

```
                FLAG the region as being 'RETAIN AFTER UPDATE'
            END IF
        END IF
        FOR j = 0 TO number of regions in RG2 DO
            comp_clab = SEARCH for an existing contents label which represents (rg1$_i$→clab comp rg2$_j$→clab).
            IF comp_clab exists AND comp_clab has SECONDARY damage but NO PRIMARY damage THEN
                inter_rgn = rg1$_i$→rgn intersection rg2$_j$→rgn
                IF inter_rgn is non-empty THEN
                    GET a reference to the existing region tagged in this region group with comp_clab which MUST exist in this region group
                    IF rg1$_i$→clab is OPAQUE THEN
                        existing regions proxy =rgl$_i$ →proxy
                    ELSE
                        update_rgn = inter_rgn difference the region's previous region description.
                        update existing regions proxy to include OVER of rgl$_i$→proxy and rg2$_j$ → proxy inside update region.
                    END IF
                    MODIFY the existing region to have inter_rgn as its region description and new_p as its proxy.
                    FLAG the region as being 'RETAIN AFTER UPDATE'
                END IF
            END IF
        END DO
    END DO

    FOR j= 0 TO number of regions in RG2 DO
        IF rg2$_j$→clab has PRIMARY or SECONDARY damage THEN
            diff_rgn = rg2$_j$→rgn difference RG1→urgn
            IF diff_rgn is non-empty THEN
                IF a region tagged with rg2$_j$→clab already exists in RG THEN
                    MODIFY it to have diff_rgn as its region description and rg2$_j$→proxy as its proxy.
            ELSE
                  ADD to RG a new region with diff_rgn as its region
description,            $rg2_j \rightarrow$proxy as its proxy and $rg2_j \rightarrow$clab as its contents
label. (*)
            END IF
            FLAG the region as being 'RETAIN AFTER UPDATE'
      END IF
   END IF
END DO

DELETE all regions of RG which are not marked RETAIN AFTER UPDATE but
whose contents labels have damage, and CLEAR flag in retained regions.
```

4.0 Tree Modifications (Linking and Unlinking)

More complex changes to a compositing tree can be achieved by changing the tree's structure. Most typical tree structure changes can be made by using two low level operations, link and unlink.

The unlink operation is used to separate a child node from its parent. After the operation is completed, the child node has no parent (meaning the child node can be linked in somewhere else), and the parent has a link available (meaning that some other node can be linked there instead). Nodes in the compositing tree above the unlinked child contain content which is dependent on the unlinked child. Therefore, at the time of the next update, the contents label present in the unlinked child at the time of unlinking must be damaged to ensure that the dependent region groups higher in the tree are appropriately updated. The updating is achieved by the parent node caching away those contents label existing in its unlinked child. If another subtree is linked in its place and subsequently unlinked without the region group of the parent being updated, it is not necessary to cache the contents labels of this new subtree. Pseudocode for the unlink operation is provided below. Note that the UNLINKED_LEFT or UNLINKED_RIGHT flag is set so that the contents labels of a newly linked subtree may be damaged when region groups (including their proxies) higher in the tree must then be updated.

```
unlink
(
   node  :  compositing tree node
)
BEGIN
    parent = node →parent.
    node →parent = NULL.
    IF node is parent's left child THEN
        parent →left = NULL.
        IF parent doesn't have UNLINKED_LEFT set THEN
            SET the UNLINKED_LEFT flag in parent.
        ELSE.
            RETURN.
        END IF
    ELSE IF node is parent's right child THEN
        parent →right = NULL.
        IF parent doesn't have UNLINKED_RIGHT set THEN
            SET the UNLINKED_RIGHT flat in parent.
        ELSE
            RETURN
        END IF
    END IF
    COPY all the contents labels in node's region group into an array stored in
parent →unlinked_clabs.
END unlink
```

The link operation involves linking a node with no parent to a free link of a parent node. Pseudocode for the operation is provided below:

```
link
(
    child  :  compositing tree node,
    parent  :  compositing tree node,
    which_link  :  either LEFT or RIGHT
)
BEGIN
    child →parent = parent
    IF which_link is LEFT THEN
```

```
        parent →left = child.
    ELSE
        parent → right = child.
    END IF
END LINK
```

4.1 Updating the Entire Compositing Tree

If a leaf node in the compositing tree changes, the region group of every node in a direct line from the leaf node to the root of tree must be updated using the methods described above. Fig. 15 shows circled those nodes which need to have their region groups updated if leaf nodes B and H change in some way.

Pseudocode for the tree updating method is provided below:

```
update_tree
(
    node : compositing tree node
)
BEGIN
    IF node is leaf node THEN
        Rerender the leaf node and update its region group.
    ELSE
        IF unlinking occurred in left subtree or left subtree contains dirty leaf
nodes THEN
            update_tree(node →left).
        END IF.
        IF unlinking occurred in right subtree or right subtree contains dirty leaf
nodes THEN
            update_tree(node →right).
        END IF.
        IF node has UNLINKED_LEFT or UNLINKED_RIGHT flags set THEN
            CALL damage_contents_label on every element of
node→unlinked_clabs.
            IF node has UNLINKED_LEFT set THEN
                CALL damage_contents_label on every contents label exist-
ing in node→left's region group.
```

```
                CLEAR the UNLINKED_LEFT flag in node.
            END IF
            IF node has UNLINKED_RIGHT set THEN
                CALL damage_contents_label on every contents label existing in node→right's region group.
                CLEAR the UNLINKED_RIGHT flag in node.
            END IF
        END IF
        CALL the region group update routine appropriate for node's compositing operator.
    END IF
END update_tree
```

The embodiments of the invention can be implemented using a conventional general-purpose computer system 2100, such as that shown in Fig. 19, wherein the process described with reference to Fig. 1 to Fig. 18 are implemented as software recorded on a computer readable medium that can be loaded into and carried out by the computer. The computer system 2100 includes a computer module 2101, input devices 2102, 2103 and a display device 2104.

With reference to Fig 19, the computer module 2101 includes at least one processor unit 2105, a memory unit 2106 which typically includes random access memory (RAM) and read only memory (ROM), input/output (I/O) interfaces including a video interface 2107, keyboard 2118 and mouse 2120 interface 2108 and an I/O interface 2110. The storage device 2109 can include one or more of the following devices: a floppy disk, a hard disk drive, a CD-ROM drive or similar a non-volatile storage device known to those skilled in the art. The components 2105 to 2110 of the computer module 2101, typically communicate via an interconnected bus 2114 and in a manner which results in a usual mode of operation of the computer system 2100 known to those in the relevant art. Examples of computer systems on which the embodiments can be practised include IBM-PC/ ATs and compatibles, Sun Sparcstations or alike computer system. In particular, the pseudocode described herein can be programmed into any appropriate language and stored for example on the HDD and executed in the RAM 2106 under control of the processor 2105 with the results being stored in RAM within the video interface 2107 and reproduced on the display 2116. The programs may be supplied to the system 2100 on a pre-programmed floppy disk or CD-ROM or accessed via a connection with a computer network, such as the Internet.

The aforementioned preferred method(s) comprise a particular control flow. There are many other variants of the preferred method(s) which use different control flows without departing the spirit or scope of the invention. Furthermore one or more of the steps of the preferred method(s) may be performed in parallel rather sequential.

The foregoing describes only several embodimens of the present invention, and modifications, obvious to those skilled in the art, can be made thereto without departing from the scope of the present invention.

In the context of this specification, the word "comprising" means "including principally but not necessarily solely" or "having" or "including" and not "consisting only of". Variations of the word comprising, such as "comprise" and "comprises" have corresponding meanings.

```
/*
 * region.cpp
 *
 * The implemention of the region manipulation functionality in
 * the Screen OpenPage prototype.
 */

#include "protos.h"

static int union_tot = 0;
static int int_tot = 0;
static int diff_tot = 0;
static int union_full = 0;
static int int_full = 0;
static int diff_full = 0;

/*
 * Some #defs which are used to control the optimisations used in the region
 * builder implementation..
 */
#define R_USE_NEW_IMP
#define RB_FAST_SHIFT_AND_DUP_LOOPS
#define RB_USE_LOOKUP
#define R_NEW_IMP_CONSTRUCTION_LOOP

/*
 * The global variables used to store the temporary results needed during
 * region manipulation operations. Two statically allocated
 * R_RegionBuilder structures are used. This is to allow data to be
 * read from one of them whilst the data required for the next operation
 * is written into the other one. Two pointers r_PrevRB and R_CurRB are
 * used to swap access to the two static structures. The
 * r_grow_region_builder function is called to grow a R_RegionBuilder
 * structure if required.
 */
static R_RegionBuilder     r_RB1 = {0, 0, NULL, NULL};
static R_RegionBuilder     r_RB2 = {0, 0, NULL, NULL};
static R_RegionBuilder     *r_PrevRB = &r_RB1;
R_RegionBuilder            *R_CurRB  = &r_RB2;

/*
 * r_shift_and_dup
 *
 * A 16-byte lookup table which when provided with an unsigned char
 * of the following form xxyy, simply produces xxxx. This lookup table
 * _assumes_ that R_STATE_SIZE is 2. It _won't_ work (and will die
 * horribly) if this isn't the case.
 */
unsigned char r_shift_and_dup[16] = {
                                        0x00, 0x00, 0x00, 0x00,
                                        0x05, 0x05, 0x05, 0x05,
                                        0x0A, 0x0A, 0x0A, 0x0A,
                                        0x0F, 0x0F, 0x0F, 0x0F
                                    };

/*
 * A buffer is required to store the new region data whilst a region is
 * being constructed. This buffer is expanded when required.
 */
static R_Int    *r_RgnBuf = NULL;
static int      r_RgnBufSize = 0;

/*
 * A buffer of IntXYMinMax structures is required to store the rectangles
 * generated during R_rects_from_region. This buffer is expanded when
 * required.
 */
static IntXYMinMax   *r_RectBuf = NULL;
static int            r_RectBufSize = 0;


/*
 * R_FREE_LIST_GROWTH_SIZE
 *
 * This macro defines the number of elements which will be added to
 * free list whenever it is grown.
 */
#define R_FREE_LIST_GROWTH_SIZE      100
/*
 * r_free_list
 *
 * A linked list of unused R_RgnGrowItems which may be used during
 * region construction.
 */
R_RgnGrowItem   *r_free_list = NULL;
/*
 * r_growth_list
 *
 * A linked list of R_RgnGrowItems which represents the current
 * state during region construction.
 */
R_RgnGrowItem   *r_growth_list = NULL;

/*
 * r_grow_region_builder
 *
 * This function simply checks to see if a R_RegionBuilder structure
 * is of the required size. If it isn't the size of both
 * arrays in the R_RegionBuilder structure are doubled.
 *
 * Parameters:
 *      rb     The region builder to be grown.
 *      size   The required size of the arrays in the R_RegionBuilder.
 * Returns:
 *      TRUE on success, FALSE on failure.
 */
static int
r_grow_region_builder
(
     R_RegionBuilder      *rb,
     R_Int                size
)
{
     unsigned char   *new_state_data;
     R_Int           *new_rgn_data;
     int             new_size;

     new_size = max(size, rb->rrb_Size * 2);
     new_state_data = (unsigned char *)malloc(new_size * sizeof(unsigned char));
     if (new_state_data == NULL)
         return FALSE;
     new_rgn_data = (R_Int *)malloc(new_size * sizeof(R_Int));
     if (new_rgn_data == NULL)
     {
         free(new_state_data);
         return FALSE;
     }
     if (rb->rrb_StateData != NULL)
     {
         memcpy
         (
                new_state_data,
                rb->rrb_StateData,
                rb->rrb_Size * sizeof(unsigned char)
         );
          free(rb->rrb_StateData);
```

```
    }
    if (rb->rrb_RgnData != NULL)
    {
        memcpy
        (
                new_rgn_data,
                    rb->rrb_RgnData,
                    rb->rrb_Size * sizeof(R_Int)
        );
        free(rb->rrb_RgnData);
    }
    rb->rrb_StateData = new_state_data;
    rb->rrb_RgnData = new_rgn_data;
    rb->rrb_Size = new_size;
    return TRUE;
}

/*
 * r_swap_region_builders
 *
 * This function simply swaps the static pointers to the r_RB1 and r_RB2
 * region builders.
 *
 * Parameters:
 *      None.
 * Returns:
 *      Nothing.
 */
inline static void
r_swap_region_builders()
{
    R_RegionBuilder *tmp;
    tmp = R_CurRB;
    R_CurRB = r_PrevRB;
    r_PrevRB = tmp;
}

/*
 * R_add_row_to_region_builder
 *
 * This function adds a row from a R_Region to a R_RegionBuilder structure.
 * The region from which the row comes is passed as an argument. "Adding"
 * has the following conditions...
 *          * If a pixel run in the row does not exist in the region builder
 *            it is added and it's current state is tagged with the region
 *            to which the row belongs. The previous state is set to 0,
 *            indicating that it did not exist before.
 *          * If a pixel run in the row did exist before, but it's present state
 *            indicates that it came from the other region then the run
 *            is retained but it's state is modified to indicate that
 *            both regions are active at this point.
 *          * If a pixel run in the row did exist before, and it's present
 *            state indicates that the current region then the region is
 *            removed and it's state is modified to indicate that the run is
 *            now empty.
 *          * If a pixel run in the row did exist before, and it's present state
 *            indicates that both regions are currently active then the run
 *            is retained, but its state is modified to indicate that only the
 *            other region is active in this run.
 *
 * Parameters:
 *          row_ptr    A R_Int ** pointer to the row in the region. Used
 *                     to return the updates row pointer.
 *          rgn_mask   A mask for the region the row comes from. Must
 *                     be either 1 or 2.
 *          first      Whether this is the first region to be processed
 *                     on the current scanline.
 * Returns:
```

```
 *         TRUE on success, FALSE on failure.
 */
#if 1
int
R_add_row_to_region_builder
(
     R_Int       **row_ptr,
     int         rgn_mask,
     int         first
)
{
     R_Int            *row;
     int              src_index;
     int              dest_index;
     R_Int            rb_run_start;
     unsigned char    rb_prev_run_state;
     int              row_on;

     ASSERT(rgn_mask == 1 || rgn_mask == 2);

     row = *row_ptr;
     r_swap_region_builders();
     /*
      * Skip over the row's y value at the beginning.
      */
     ASSERT(*row == R_NEXT_IS_Y);
     row += 2;
     ASSERT(*row != R_NEXT_IS_Y && *row != R_EOR);

     if (r_PrevRB->rrb_Nels == 0)
     {
          /*
           * If the current region builder's src region data array is empty, then
           * we are dealing with an empty region builder. We simply convert
           * the input row to the region builder format.
           */
          row_on = TRUE;
          dest_index = 0;
          while (R_NOT_END_OF_ROW(*row))
          {
               if (++dest_index > R_CurRB->rrb_Size)
               {
                    if (!r_grow_region_builder(R_CurRB, dest_index))
                         return FALSE;
               }
               R_CurRB->rrb_RgnData[dest_index - 1] = *row;
               if (row_on)
                    R_CurRB->rrb_StateData[dest_index - 1] =
                                                       (rgn_mask << RB_STATE_SIZE);
               else
                    R_CurRB->rrb_StateData[dest_index - 1] = 0;
               row_on = !row_on;
               row++;
          }
          *row_ptr = row;
          R_CurRB->rrb_Nels = dest_index;
          return TRUE;
     }
     /*
      * Firstly, we copy any runs from the region builder which
      * precede this run from the region. We are checking the
      * starting row against the start of each pixel run. Therefore
      * we start checking against the 1nd region builder data
      * element.
      */
     ASSERT(r_PrevRB->rrb_Nels >= 2);
     src_index = 0;
     while (src_index < r_PrevRB->rrb_Nels && *row > r_PrevRB->rrb_RgnData[src_index])
```

```
            src_index++;
        dest_index = src_index;
        if (src_index > 0)
        {
            if (src_index > R_CurRB->rrb_Size)
            {
                if (!r_grow_region_builder(R_CurRB, src_index))
                    return FALSE;
            }
            memcpy
            (
                R_CurRB->rrb_RgnData,
                r_PrevRB->rrb_RgnData,
                src_index * sizeof(R_Int)
            );
            if (!first)
            {
                memcpy
                (
                    R_CurRB->rrb_StateData,
                    r_PrevRB->rrb_StateData,
                    src_index * sizeof(unsigned char)
                );
            }
            else
            {
                int             i = 0;
                unsigned char   *src;
                unsigned char   *dest;

                src = r_PrevRB->rrb_StateData + src_index;
                dest = R_CurRB->rrb_StateData + src_index;
                switch (src_index)
                {
                default:
                    for (i = src_index; i > 10; i--)
                    {
#ifndef RB_USE_LOOKUP
                        *(dest - i) =   (*(src - i) & RB_CUR_STATE_MASK) |
                                        (*(src - i) >> RB_STATE_SIZE);
#else
                        *(dest - i) =   r_shift_and_dup[*(src - i)];
#endif
                    }
                    /* FALLTHROUGH!! */
                case 10:
#ifndef RB_USE_LOOKUP
                    *(dest - 10) =  (*(src - 10) & RB_CUR_STATE_MASK) |
                                    (*(src - 10) >> RB_STATE_SIZE);
#else
                    *(dest - 10) =  r_shift_and_dup[*(src - 10)];
#endif
                    /* FALLTHROUGH!! */
                case 9:
#ifndef RB_USE_LOOKUP
                    *(dest - 9) =   (*(src - 9) & RB_CUR_STATE_MASK) |
                                    (*(src - 9) >> RB_STATE_SIZE);
#else
                    *(dest - 9) =   r_shift_and_dup[*(src - 9)];
#endif
                    /* FALLTHROUGH!! */
                case 8:
#ifndef RB_USE_LOOKUP
                    *(dest - 8) =   (*(src - 8) & RB_CUR_STATE_MASK) |
                                    (*(src - 8) >> RB_STATE_SIZE);
#else
                    *(dest - 8) =   r_shift_and_dup[*(src - 8)];
#endif
```

```
                    /* FALLTHROUGH!! */
               case 7:
#ifndef RB_USE_LOOKUP
                    *(dest - 7) =   (*(src - 7) & RB_CUR_STATE_MASK) |
                                    (*(src - 7) >> RB_STATE_SIZE);
#else
                    *(dest - 7) =   r_shift_and_dup[*(src - 7)];
#endif
                    /* FALLTHROUGH!! */
               case 6:
#ifndef R_USE_LOOKUP
                    *(dest - 6) =   (*(src - 6) & RB_CUR_STATE_MASK) |
                                    (*(src - 6) >> RB_STATE_SIZE);
#else
                    *(dest - 6) =   r_shift_and_dup[*(src - 6)];
#endif
                    /* FALLTHROUGH!! */
               case 5:
#ifndef RB_USE_LOOKUP
                    *(dest - 5) =   (*(src - 5) & RB_CUR_STATE_MASK) |
                                    (*(src - 5) >> RB_STATE_SIZE);
#else
                    *(dest - 5) =   r_shift_and_dup[*(src - 5)];
#endif
                    /* FALLTHROUGH!! */
               case 4:
#ifndef RB_USE_LOOKUP
                    *(dest - 4) =   (*(src - 4) & RB_CUR_STATE_MASK) |
                                    (*(src - 4) >> RB_STATE_SIZE);
#else
                    *(dest - 4) =   r_shift_and_dup[*(src - 4)];
#endif
                    /* FALLTHROUGH!! */
               case 3:
#ifndef RB_USE_LOOKUP
                    *(dest - 3) =   (*(src - 3) & RB_CUR_STATE_MASK) |
                                    (*(src - 3) >> RB_STATE_SIZE);
#else
                    *(dest - 3) =   r_shift_and_dup[*(src - 3)];
#endif
                    /* FALLTHROUGH!! */
               case 2:
#ifndef RB_USE_LOOKUP
                    *(dest - 2) =   (*(src - 2) & RB_CUR_STATE_MASK) |
                                    (*(src - 2) >> RB_STATE_SIZE);
#else
                    *(dest - 2) =   r_shift_and_dup[*(src - 2)];
#endif
                    /* FALLTHROUGH!! */
               case 1:
#ifndef RB_USE_LOOKUP
                    *(dest - 1) =   (*(src - 1) & RB_CUR_STATE_MASK) |
                                    (*(src - 1) >> RB_STATE_SIZE);
#else
                    *(dest - 1) =   r_shift_and_dup[*(src - 1)];
#endif
                    /* FALLTHROUGH!! */
               case 0:
                    ;
                    /* FALLTHROUGH!! */
               }
          }
     }
     if   (src_index == r_PrevRB->rrb_Nels)
     {
          /*
           * We've already exhausted the previous region builder. Set the start
           * of the next pixel run to be the max. possible and set the state
```

```
         * to be 0.
         */
        rb_run_start = R_INT_MAX_VALUE - 2;
        rb_prev_run_state = 0;
    }
    else
    {
        /*
         * We are still within the previous region builder bounds. Set up
         * the run info appropriately.
         */
        rb_run_start = r_PrevRB->rrb_RgnData[src_index];
        if (src_index == 0)
            rb_prev_run_state = 0;
        else
            rb_prev_run_state = r_PrevRB->rrb_StateData[src_index - 1];
    }

    /*
     * We can now start dealing with the elements in the row.
     */
    row_on = 1;
    while (R_NOT_END_OF_ROW(*row))
    {
        if (*row < rb_run_start)
        {
            if (dest_index + 1 > R_CurRB->rrb_Size)
            {
                if (!r_grow_region_builder(R_CurRB, dest_index + 1))
                    return FALSE;
            }
            R_CurRB->rrb_RgnData[dest_index] = *row;
            if (first)
            {
                /*
                 * We are processing the first region. Therefore, we
                 * copy the current state of the run to the lowest
                 * RB_STATE_SIZE bits.
                 */
                R_CurRB->rrb_StateData[dest_index] =
                                (rb_prev_run_state & RB_CUR_STATE_MASK) |
                                (rb_prev_run_state >> RB_STATE_SIZE);
            }
            else
            {
                /*
                 * We are processing the second region. Therefore, the state data
                 * has already been copied to the previous state area so we
                 * just copy the state.
                 */
                R_CurRB->rrb_StateData[dest_index] = rb_prev_run_state;
            }
            /*
             * Now, if the row for the current region is active at this transition,
             * we xor the region mask with the current contents of the new region
             * builder slot. This gives the desired behaviour of making that region
             * active if it is not there already, but turns it off if it is...
             */
            if (row_on)
                R_CurRB->rrb_StateData[dest_index] ^= (rgn_mask << RB_STATE_SIZE);
            dest_index++;
            /*
             * We now move onto the next row element.
             */
            row++;
            row_on = !row_on;
            continue;
        }
        /*
         * If the current row transition point is equal in x position to the current
         * previous region builder transition point, we advance the row counter to
         * the next position.
         */
        if (*row == rb_run_start)
        {
            row++;
            row_on = !row_on;
        }
        /*
         * Output the previous regions builder's transition region. We do similiar
         * things as for the region transition stuff above.. Firstly though, we
         * advance the rb_prev_run_state variable to the next element. We know
         * we can do this because if we were on the last element, we wouldn't
         * have hit this section of code.
         */
        rb_prev_run_state = r_PrevRB->rrb_StateData[src_index];
        if (dest_index + 1 > R_CurRB->rrb_Size)
        {
            if (!r_grow_region_builder(R_CurRB, dest_index + 1))
                return FALSE;
        }
        R_CurRB->rrb_RgnData[dest_index] = rb_run_start;
        if (first)
        {
            R_CurRB->rrb_StateData[dest_index]=(rb_prev_run_state &
                                                RB_CUR_STATE_MASK) |
                                                rb_prev_run_state >> RB_STATE_SIZE);
        }
        else
        {
            R_CurRB->rrb_StateData[dest_index] = rb_prev_run_state;
        }
        if (!row_on)
            R_CurRB->rrb_StateData[dest_index] ^=  (rgn_mask << RB_STATE_SIZE);
        dest_index++;
        /*
         * We've output the previous region builder's transitions. We now move
         * over onto the next transition. If the previous src_index increment
         * has moved us onto the last element, we declare that we have run
         * out of previous region builder data.
         */
        ASSERT(rb_run_start != R_EOR);
        if (++src_index >= r_PrevRB->rrb_Nels)
        {
            /*
             * We've run out of data..
             */
            rb_run_start = R_INT_MAX_VALUE - 2;
            continue;
        }
        /*
         * Otherwise, we still have stuff left to do, so we move onto
         * the next run in the previous region builder.
         */
        rb_run_start = r_PrevRB->rrb_RgnData[src_index];
    }
    /*
     * Now, we simply blast out any remaining region builder transition
     * points.
     */
    if (r_PrevRB->rrb_Nels - src_index > 0)
    {
        R_Intnels_to_copy;
        nels_to_copy = r_PrevRB->rrb_Nels - src_index;
        if (dest_index + nels_to_copy > R_CurRB->rrb_Size)
        {
                if (!r_grow_region_builder(R_CurRB, dest_index + nels_to_copy))
                    return FALSE;
            }

            memcpy
            (
                R_CurRB->rrb_RgnData + dest_index,
                r_PrevRB->rrb_RgnData + src_index,
                nels_to_copy * sizeof(R_Int)
            );
            if (!first)
            {
                memcpy
                (
                    R_CurRB->rrb_StateData + dest_index,
                    r_PrevRB->rrb_StateData + src_index,
                    nels_to_copy * sizeof(unsigned char)
                );
                dest_index += nels_to_copy;
            }
            else
            {
                int             i = 0;
                unsigned char   *src;
                unsigned char   *dest;

                i = r_PrevRB->rrb_Nels - src_index;
                src = r_PrevRB->rrb_StateData + r_PrevRB->rrb_Nels;
                dest_index += i;
                dest = R_CurRB->rrb_StateData + dest_index;
                switch (i)
                {
                default:
                    for (; i > 10; i--)
                    {
#ifndef RB_USE_LOOKUP
                        *(dest - i) =   (*(src - i) & RB_CUR_STATE_MASK) |
                                        (*(src - i) >> RB_STATE_SIZE);
#else
                        *(dest - i) =   r_shift_and_dup[*(src - i)];
#endif
                    }
                    /* FALLTHROUGH!! */
                case 10:
#ifndef RB_USE_LOOKUP
                    *(dest - 10) =  (*(src - 10) & RB_CUR_STATE_MASK) |
                                    (*(src - 10) >> RB_STATE_SIZE);
#else
                    *(dest - 10) =  r_shift_and_dup[*(src - 10)];
#endif
                    /* FALLTHROUGH!! */
                case 9:
#ifndef RB_USE_LOOKUP
                    *(dest - 9) =   (*(src - 9) & RB_CUR_STATE_MASK) |
                                    (*(src - 9) >> RB_STATE_SIZE);
#else
                    *(dest - 9) =   r_shift_and_dup[*(src - 9)];
#endif
                    /* FALLTHROUGH!! */
                case 8:
#ifndef RB_USE_LOOKUP
                    *(dest - 8) =   (*(src - 8) & RB_CUR_STATE_MASK) |
                                    (*(src - 8) >> RB_STATE_SIZE);
#else
                    *(dest - 8) =   r_shift_and_dup[*(src - 8)];
#endif
                    /* FALLTHROUGH!! */
                case 7:
```

```
#ifndef RB_USE_LOOKUP
                        *(dest - 7) =   (*(src - 7) & RB_CUR_STATE_MASK) |
                                        (*(src - 7) >> RB_STATE_SIZE);
#else
                        *(dest - 7) =   r_shift_and_dup[*(src - 7)];
#endif
                        /* FALLTHROUGH!! */
                    case 6:
#ifndef RB_USE_LOOKUP
                        *(dest - 6) =   (*(src - 6) & RB_CUR_STATE_MASK) |
                                        (*(src - 6) >> RB_STATE_SIZE);
#else
                        *(dest - 6) =   r_shift_and_dup[*(src - 6)];
#endif
                        /* FALLTHROUGH!! */
                    case 5:
#ifndef RB_USE_LOOKUP
                        *(dest - 5) =   (*(src - 5) & RB_CUR_STATE_MASK) |
                                        (*(src - 5) >> RB_STATE_SIZE);
#else
                        *(dest - 5) =   r_shift_and_dup[*(src - 5)];
#endif
                        /* FALLTHROUGH!! */
                    case 4:
#ifndef RB_USE_LOOKUP
                        *(dest - 4) =   (*(src - 4) & RB_CUR_STATE_MASK) |
                                        (*(src - 4) >> RB_STATE_SIZE);
#else
                        *(dest - 4) =   r_shift_and_dup[*(src - 4)];
#endif
                        /* FALLTHROUGH!! */
                    case 3:
#ifndef RB_USE_LOOKUP
                        *(dest - 3) =   (*(src - 3) & RB_CUR_STATE_MASK) |
                                        (*(src - 3) >> RB_STATE_SIZE);
#else
                        *(dest - 3) =   r_shift_and_dup[*(src - 3)];
#endif
                        /* FALLTHROUGH!! */
                    case 2:
#ifndef RB_USE_LOOKUP
                        *(dest - 2) =   (*(src - 2) & RB_CUR_STATE_MASK) |
                                        (*(src - 2) >> RB_STATE_SIZE);
#else
                        *(dest - 2) =   r_shift_and_dup[*(src - 2)];
#endif
                        /* FALLTHROUGH!! */
                    case 1:
#ifndef RB_USE_LOOKUP
                        *(dest - 1) =   (*(src - 1) & RB_CUR_STATE_MASK) |
                                        (*(src - 1) >> RB_STATE_SIZE);
#else
                        *(dest - 1) =   r_shift_and_dup[*(src - 1)];
#endif
                        /* FALLTHROUGH!! */
                    case 0:
                        ;
                        /* FALLTHROUGH!! */
                }
            }
        }
        /*
         * Finally, we set the number of elements of the latest region
         * builder. We also return the updates row variable.
         */
        R_CurRB->rrb_Nels = dest_index;
        *row_ptr = row;
        return TRUE;
```

```
}
#else
int
R_add_row_to_region_builder
(
     R_Int      **row_ptr,
     int        rgn_mask,
     int        first
)
{
     R_Int            *row;
     R_Int            rb_run_start;
     unsigned char    rb_prev_run_state;
     int              row_on;

     int              dest_index;
     unsigned char    *src_state_ptr;
     unsigned char    *dest_state_ptr;
     unsigned char    *src_state_end_ptr;
     register R_Int             *src_rgn_ptr;
     R_Int            *src_rgn_end_ptr;
     register R_Int             *dest_rgn_ptr;
     R_Int            *dest_rgn_end_ptr;
     int              inc;
     int              i;

     ASSERT(rgn_mask == 1 || rgn_mask == 2);

     row = *row_ptr;
     r_swap_region_builders();
     /*
      * Skip over the row's y value at the beginning.
      */
     ASSERT(*row == R_NEXT_IS_Y);
     row += 2;
     ASSERT(*row != R_NEXT_IS_Y && *row != R_EOR);

     if (r_PrevRB->rrb_Nels == 0)
     {
          /*
           * If the current region builder's src region data array is empty, then
           * we are dealing with an empty region builder. We simply convert
           * the input row to the region builder format.
           */
          row_on = TRUE;
          dest_index = 0;
          while (R_NOT_END_OF_ROW(*row))
          {
               if (++dest_index > R_CurRB->rrb_Size)
               {
                    if (!r_grow_region_builder(R_CurRB, dest_index))
                        return FALSE;
               }
               R_CurRB->rrb_RgnData[dest_index - 1] = *row;
               if (row_on)
                    R_CurRB->rrb_StateData[dest_index - 1] =
                                                (rgn_mask << RB_STATE_SIZE);
               else
                    R_CurRB->rrb_StateData[dest_index - 1] = 0;
               row_on = !row_on;
               row++;
          }
          *row_ptr = row;
          R_CurRB->rrb_Nels = dest_index;
          return TRUE;
     }
     /*
      * Firstly, we copy any runs from the region builder which
```

```
         * precede this run from the region. We are checking the
         * starting row against the start of each pixel run. Therefore
         * we start checking against the 1nd region builder data
         * element.
         */
        src_state_ptr = r_PrevRB->rrb_StateData;
        src_rgn_ptr = r_PrevRB->rrb_RgnData;
        src_state_end_ptr = src_state_ptr + r_PrevRB->rrb_Nels;
        src_rgn_end_ptr = src_rgn_ptr + r_PrevRB->rrb_Nels;
        dest_state_ptr = R_CurRB->rrb_StateData;
        dest_rgn_ptr = R_CurRB->rrb_RgnData;
        dest_rgn_end_ptr = dest_rgn_ptr + R_CurRB->rrb_Size;

        ASSERT(r_PrevRB->rrb_Nels >= 2);
        while (src_rgn_ptr != src_rgn_end_ptr && *row > *src_rgn_ptr)
             src_rgn_ptr++;
        inc = src_rgn_ptr - r_PrevRB->rrb_RgnData;

        if (inc > 0)
        {
             src_state_ptr += inc;
             dest_state_ptr += inc;
             dest_rgn_ptr += inc;
             if (dest_rgn_ptr >  dest_rgn_end_ptr)
             {
                  if (!r_grow_region_builder(R_CurRB, inc))
                       return FALSE;
                  dest_state_ptr = R_CurRB->rrb_StateData;
                  dest_rgn_ptr = R_CurRB->rrb_RgnData;
                  dest_rgn_end_ptr = dest_rgn_ptr + R_CurRB->rrb_Size;
             }
#if 1
             const R_Int      * const src_rgn_ptr2 = src_rgn_ptr;
             R_Int            *dest_rgn_ptr2 = dest_rgn_ptr;
             switch(inc)
             {
             default:
                  for (i = inc; i > 10; i--)
                  {
                       *(dest_rgn_ptr2 - i) = *(src_rgn_ptr - i);
                  }
                  /* FALLTHROUGH!! */
             case 10:
                  *(dest_rgn_ptr2 - 10) = *(src_rgn_ptr2 - 10);
                  /* FALLTHROUGH!! */
             case 9:
                  *(dest_rgn_ptr2 - 9) = *(src_rgn_ptr2 - 9);
                  /* FALLTHROUGH!! */
             case 8:
                  *(dest_rgn_ptr2 - 8) = *(src_rgn_ptr2 - 8);
                  /* FALLTHROUGH!! */
             case 7:
                  *(dest_rgn_ptr2 - 7) = *(src_rgn_ptr2 - 7);
                  /* FALLTHROUGH!! */
             case 6:
                  *(dest_rgn_ptr2 - 6) = *(src_rgn_ptr2 - 6);
                  /* FALLTHROUGH!! */
             case 5:
                  *(dest_rgn_ptr2 - 5) = *(src_rgn_ptr2 - 5);
                  /* FALLTHROUGH!! */
             case 4:
                  *(dest_rgn_ptr2 - 4) = *(src_rgn_ptr2 - 4);
                  /* FALLTHROUGH!! */
             case 3:
                  *(dest_rgn_ptr2 - 3) = *(src_rgn_ptr2 - 3);
                  /* FALLTHROUGH!! */
             case 2:
                  *(dest_rgn_ptr2 - 2) = *(src_rgn_ptr2 - 2);
```

```
                /* FALLTHROUGH!! */
            case 1:
                *(dest_rgn_ptr2 - 1) = *(src_rgn_ptr2 - 1);
                /* FALLTHROUGH!! */
            case 0:
                ;
                /* FALLTHROUGH!! */
            }
#else
            memcpy
            (
                R_CurRB->rrb_RgnData,
                r_PrevRB->rrb_RgnData,
                inc * sizeof(R_Int)
            );
#endif
            if (!first)
            {
                memcpy
                (
                    R_CurRB->rrb_StateData,
                    r_PrevRB->rrb_StateData,
                    inc * sizeof(unsigned char)
                );
            }
            else
            {
                switch (inc)
                {
                default:
                    for (i = inc; i > 10; i--)
                    {
#ifndef RB_USE_LOOKUP
                        *(dest_state_ptr - i) = (*(src_state_ptr - i) &
                                                 RB_CUR_STATE_MASK) |
                                                (*(src_state_ptr - i) >>
                                                 RB_STATE_SIZE);
#else
                        *(dest_state_ptr - i) = r_shift_and_dup[*(src_state_ptr - i)];
#endif
                    }
                    /* FALLTHROUGH!! */
                case 10:
#ifndef RB_USE_LOOKUP
                    *(dest_state_ptr - 10) = (*(src_state_ptr - 10) &
                                              RB_CUR_STATE_MASK) |
                                             (*(src_state_ptr - 10) >>
                                              RB_STATE_SIZE);
#else
                    *(dest_state_ptr - 10) = r_shift_and_dup[*(src_state_ptr - 10)];
#endif
                    /* FALLTHROUGH!! */
                case 9:
#ifndef RB_USE_LOOKUP
                    *(dest_state_ptr - 9)=(*(src_state_ptr - 9) & RB_CUR_STATE_MASK) |
                                          (*(src_state_ptr - 9) >> RB_STATE_SIZE);
#else
                    *(dest_state_ptr - 9) = r_shift_and_dup[*(src_state_ptr - 9)];
#endif
                    /* FALLTHROUGH!! */
                case 8:
#ifndef RB_USE_LOOKUP
                    *(dest_state_ptr - 8)=(*(src_state_ptr - 8) & RB_CUR_STATE_MASK) |
                                          (*(src_state_ptr - 8) >> RB_STATE_SIZE);
#else
                    *(dest_state_ptr - 8) = r_shift_and_dup[*(src_state_ptr - 8)];
#endif
                    /* FALLTHROUGH!! */
                case 7:
#ifndef RB_USE_LOOKUP
                     *(dest_state_ptr - 7)= (*(src_state_ptr - 7) & RB_CUR_STATE_MASK) |
                                            (*(src_state_ptr - 7) >> RB_STATE_SIZE);
#else
                     *(dest_state_ptr - 7) = r_shift_and_dup[*(src_state_ptr - 7)];
#endif
                     /* FALLTHROUGH!! */
                case 6:
#ifndef R_USE_LOOKUP
                     *(dest_state_ptr - 6)= (*(src_state_ptr - 6) & RB_CUR_STATE_MASK) |
                                            (*(src_state_ptr - 6) >> RB_STATE_SIZE);
#else
                     *(dest_state_ptr - 6) = r_shift_and_dup[*(src_state_ptr - 6)];
#endif
                     /* FALLTHROUGH!! */
                case 5:
#ifndef RB_USE_LOOKUP
                     *(dest_state_ptr - 5)= (*(src_state_ptr - 5) & RB_CUR_STATE_MASK) |
                                            (*(src_state_ptr - 5) >> RB_STATE_SIZE);
#else
                     *(dest_state_ptr - 5) = r_shift_and_dup[*(src_state_ptr - 5)];
#endif
                     /* FALLTHROUGH!! */
                case 4:
#ifndef RB_USE_LOOKUP
                     *(dest_state_ptr - 4)= (*(src_state_ptr - 4) & RB_CUR_STATE_MASK) |
                                            (*(src_state_ptr - 4) >> RB_STATE_SIZE);
#else
                     *(dest_state_ptr - 4) = r_shift_and_dup[*(src_state_ptr - 4)];
#endif
                     /* FALLTHROUGH!! */
                case 3:
#ifndef RB_USE_LOOKUP
                     *(dest_state_ptr - 3)= (*(src_state_ptr - 3) & RB_CUR_STATE_MASK) |
                                            (*(src_state_ptr - 3) >> RB_STATE_SIZE);
#else
                     *(dest_state_ptr - 3) = r_shift_and_dup[*(src_state_ptr - 3)];
#endif
                     /* FALLTHROUGH!! */
                case 2:
#ifndef RB_USE_LOOKUP
                     *(dest_state_ptr - 2)= (*(src_state_ptr - 2) & RB_CUR_STATE_MASK) |
                                            (*(src_state_ptr - 2) >> RB_STATE_SIZE);
#else
                     *(dest_state_ptr - 2) = r_shift_and_dup[*(src_state_ptr - 2)];
#endif
                     /* FALLTHROUGH!! */
                case 1:
#ifndef RB_USE_LOOKUP
                     *(dest_state_ptr - 1)= (*(src_state_ptr - 1) & RB_CUR_STATE_MASK) |
                                            (*(src_state_ptr - 1) >> RB_STATE_SIZE);
#else
                     *(dest_state_ptr - 1) = r_shift_and_dup[*(src_state_ptr - 1)];
#endif
                     /* FALLTHROUGH!! */
                case 0:
                     ;
                     /* FALLTHROUGH!! */
                }
           }
     }
     if (src_state_ptr == src_state_end_ptr)
     {
           /*
            * We've already exhausted the previous region builder. Set the start
            * of the next pixel run to be the max. possible and set the state
            * to be 0.
```

```
         */
        rb_run_start = R_INT_MAX_VALUE - 2;
        rb_prev_run_state = 0;
    }
    else
    {
        /*
         * We are still within the previous region builder bounds. Set up
         * the run info appropriately.
         */
        rb_run_start = *src_rgn_ptr;
        if (src_state_ptr == r_PrevRB->rrb_StateData)
            rb_prev_run_state = 0;
        else
            rb_prev_run_state = *(src_state_ptr - 1);
    }

    /*
     * We can now start dealing with the elements in the row.
     */
    row_on = 1;
    while (R_NOT_END_OF_ROW(*row))
    {
        if (*row < rb_run_start)
        {
            if (dest_rgn_ptr + 1 > dest_rgn_end_ptr)
            {
                if (!r_grow_region_builder(R_CurRB, dest_rgn_ptr - R_CurRB-
                    >rrb_RgnData + 1)) return FALSE;
                dest_state_ptr = R_CurRB->rrb_StateData;
                dest_rgn_ptr = R_CurRB->rrb_RgnData;
                dest_rgn_end_ptr = dest_rgn_ptr + R_CurRB->rrb_Size;
            }
            *dest_rgn_ptr = *row;
            if (first)
            {
                /*
                 * We are processing the first region. Therefore, we
                 * copy the current state of the run to the lowest
                 * RB_STATE_SIZE bits.
                 */
                *dest_state_ptr = (rb_prev_run_state & RB_CUR_STATE_MASK) |
                                  (rb_prev_run_state >> RB_STATE_SIZE);
            }
            else
            {
                /*
                 * We are processing the second region. Therefore, the state data
                 * has already been copied to the previous state area so we
                 * just copy the state.
                 */
                *dest_state_ptr = rb_prev_run_state;
            }
            /*
             * Now, if the row for the current region is active at this transition,
             * we xor the region mask with the current contents of the new region
             * builder slot. This gives the desired behaviour of making that region
             * active if it is not there already, but turns it off if it is...
             */
            if (row_on)
                *dest_state_ptr ^= (rgn_mask << RB_STATE_SIZE);
            dest_state_ptr++;
            dest_rgn_ptr++;
            /*
             * We now move onto the next row element.
             */
            row++;
            row_on = !row_on;
```

```
                continue;
          }
          /*
           * If the current row transition point is equal in x position to the current
           * previous region builder transition point, we advance the row counter to
           * the next position.
           */
          if (*row == rb_run_start)
          {
                row++;
                row_on = !row_on;
          }
          /*
           * Output the previous regions builder's transition region. We do similiar
           * things as for the region transition stuff above.. Firstly though, we
           * advance the rb_prev_run_state variable to the next element. We know
           * we can do this because if we were on the last element, we wouldn't
           * have hit this section of code.
           */
          rb_prev_run_state = *src_state_ptr;
          if (dest_rgn_ptr + 1 > dest_rgn_end_ptr)
          {
                if (!r_grow_region_builder(R_CurRB, dest_rgn_ptr - R_CurRB->rrb_RgnData
                    + 1)) return FALSE;
                dest_state_ptr = R_CurRB->rrb_StateData;
                dest_rgn_ptr = R_CurRB->rrb_RgnData;
                dest_rgn_end_ptr = dest_rgn_ptr + R_CurRB->rrb_Size;
          }
          *dest_rgn_ptr = rb_run_start;
          if (first)
          {
                *dest_state_ptr = (rb_prev_run_state & RB_CUR_STATE_MASK) |
                                  (rb_prev_run_state >> RB_STATE_SIZE);
          }
          else
          {
                *dest_state_ptr = rb_prev_run_state;
          }
          if (!row_on)
                *dest_state_ptr ^=  (rgn_mask << RB_STATE_SIZE);
          dest_state_ptr++;
          dest_rgn_ptr++;
          /*
           * We've output the previous region builder's transitions. We now move
           * over onto the next transition. If the previous src_index increment
           * has moved us onto the last element, we declare that we have run
           * out of previous region builder data.
           */
          ASSERT(rb_run_start != R_EOR);
          ++src_rgn_ptr;
          if (++src_state_ptr == src_state_end_ptr)
          {
                /*
                 * We've run out of data..
                 */
                rb_run_start = R_INT_MAX_VALUE - 2;
                continue;
          }
          /*
           * Otherwise, we still have stuff left to do, so we move onto
           * the next run in the previous region builder.
           */
          rb_run_start = *src_rgn_ptr;
    }
    /*
     * Now, we simply blast out any remaining region builder transition
     * points.
     */
    if (src_state_ptr != src_state_end_ptr)
    {
        R_Int nels_to_copy;
        nels_to_copy = src_state_end_ptr - src_state_ptr;
        if (dest_rgn_ptr + nels_to_copy > dest_rgn_end_ptr)
        {
            if
            (
                !r_grow_region_builder
                (
                    R_CurRB,
                    dest_rgn_ptr - R_CurRB->rrb_RgnData + nels_to_copy
                )
            )
                return FALSE;
            dest_state_ptr = R_CurRB->rrb_StateData;
            dest_rgn_ptr = R_CurRB->rrb_RgnData;
        }
        memcpy
        (
            dest_rgn_ptr,
            src_rgn_ptr,
            nels_to_copy * sizeof(R_Int)
        );
        dest_rgn_ptr += nels_to_copy;
        if (!first)
        {
            memcpy
            (
                dest_state_ptr,
                src_state_ptr,
                nels_to_copy * sizeof(unsigned char)
            );
        }
        else
        {
            i = nels_to_copy;
            src_state_ptr = src_state_end_ptr;
            dest_state_ptr += nels_to_copy;
            switch (i)
            {
            default:
                for (; i > 10; i--)
                {
#ifndef RB_USE_LOOKUP
                    *(dest_state_ptr - i)= (*(src_state_ptr - i) & RB_CUR_STATE_MASK) |
                                          (*(src_state_ptr - i) >> RB_STATE_SIZE);
#else
                    *(dest_state_ptr - i) = r_shift_and_dup[*(src_state_ptr - i)];
#endif
                }
                /* FALLTHROUGH!! */
            case 10:
#ifndef RB_USE_LOOKUP
                    *(dest_state_ptr - 10)= (*(src_state_ptr - 10)&RB_CUR_STATE_MASK) |
                                            (*(src_state_ptr - 10) >> RB_STATE_SIZE);
#else
                    *(dest_state_ptr - 10) = r_shift_and_dup[*(src_state_ptr - 10)];
#endif
                    /* FALLTHROUGH!! */
            case 9:
#ifndef RB_USE_LOOKUP
                    *(dest_state_ptr - 9)= (*(src_state_ptr - 9) & RB_CUR_STATE_MASK) |
                                          (*(src_state_ptr - 9) >> RB_STATE_SIZE);
#else
                    *(dest_state_ptr - 9) = r_shift_and_dup[*(src_state_ptr - 9)];
#endif
                    /* FALLTHROUGH!! */
                case 8:
#ifndef RB_USE_LOOKUP
                      *(dest_state_ptr - 8)= (*(src_state_ptr - 8) & RB_CUR_STATE_MASK) |
                                             (*(src_state_ptr - 8) >> RB_STATE_SIZE);
#else
                      *(dest_state_ptr - 8) = r_shift_and_dup[*(src_state_ptr - 8)];
#endif
                      /* FALLTHROUGH!! */
                case 7:
#ifndef RB_USE_LOOKUP
                      *(dest_state_ptr - 7)= (*(src_state_ptr - 7) & RB_CUR_STATE_MASK) |
                                             (*(src_state_ptr - 7) >> RB_STATE_SIZE);
#else
                      *(dest_state_ptr - 7) = r_shift_and_dup[*(src_state_ptr - 7)];
#endif
                      /* FALLTHROUGH!! */
                case 6:
#ifndef R_USE_LOOKUP
                      *(dest_state_ptr - 6)= (*(src_state_ptr - 6) & RB_CUR_STATE_MASK) |
                                             (*(src_state_ptr - 6) >> RB_STATE_SIZE);
#else
                      *(dest_state_ptr - 6) = r_shift_and_dup[*(src_state_ptr - 6)];
#endif
                      /* FALLTHROUGH!! */
                case 5:
#ifndef RB_USE_LOOKUP
                      *(dest_state_ptr - 5)= (*(src_state_ptr - 5) & RB_CUR_STATE_MASK) |
                                             (*(src_state_ptr - 5) >> RB_STATE_SIZE);
#else
                      *(dest_state_ptr - 5) = r_shift_and_dup[*(src_state_ptr - 5)];
#endif
                      /* FALLTHROUGH!! */
                case 4:
#ifndef RB_USE_LOOKUP
                      *(dest_state_ptr - 4)= (*(src_state_ptr - 4) & RB_CUR_STATE_MASK) |
                                             (*(src_state_ptr - 4) >> RB_STATE_SIZE);
#else
                      *(dest_state_ptr - 4) = r_shift_and_dup[*(src_state_ptr - 4)];
#endif
                      /* FALLTHROUGH!! */
                case 3:
#ifndef RB_USE_LOOKUP
                      *(dest_state_ptr - 3)= (*(src_state_ptr - 3) & RB_CUR_STATE_MASK) |
                                             (*(src_state_ptr - 3) >> RB_STATE_SIZE);
#else
                      *(dest_state_ptr - 3) = r_shift_and_dup[*(src_state_ptr - 3)];
#endif
                      /* FALLTHROUGH!! */
                case 2:
#ifndef RB_USE_LOOKUP
                      *(dest_state_ptr - 2)= (*(src_state_ptr - 2) & RB_CUR_STATE_MASK) |
                                             (*(src_state_ptr - 2) >> RB_STATE_SIZE);
#else
                      *(dest_state_ptr - 2) = r_shift_and_dup[*(src_state_ptr - 2)];
#endif
                      /* FALLTHROUGH!! */
                case 1:
#ifndef RB_USE_LOOKUP
                      *(dest_state_ptr - 1)= (*(src_state_ptr - 1) & RB_CUR_STATE_MASK) |
                                             (*(src_state_ptr - 1) >> RB_STATE_SIZE);
#else
                      *(dest_state_ptr - 1) = r_shift_and_dup[*(src_state_ptr - 1)];
#endif
                      /* FALLTHROUGH!! */
                case 0:
                      ;
                      /* FALLTHROUGH!! */
                }
```

```
          }
     }
     /*
      * Finally, we set the number of elements of the latest region
      * builder. We also return the updates row variable.
      */
     R_CurRB->rrb_Nels = dest_rgn_ptr - R_CurRB->rrb_RgnData;
     *row_ptr = row;
     return TRUE;
}
#endif

/*
 * r_check_rgn_buf_len
 *
 * This function checks to see if the static region buffer is large enough.
 * If it isn't then it is reallocated to make it large enough.
 *
 * Parameters:
 *        size The required size of the r_RegBuf array.
 * Returns:
 *        TRUE on success, FALSE on failure.
 */
static int
r_check_rgn_buf_len
(
     int        size
)
{
     ASSERT(size >= 0);
     if (size > r_RgnBufSize)
     {
          int        new_buf_size;
          R_Int      *new_buf;
          new_buf_size = max(size, r_RgnBufSize * 2);
          new_buf = (R_Int *)malloc(new_buf_size * sizeof(R_Int));
          if (new_buf == NULL)
               return FALSE;
          if (r_RgnBuf != NULL)
          {
               memcpy(new_buf, r_RgnBuf, r_RgnBufSize * sizeof(R_Int));
               free(r_RgnBuf);
          }
          r_RgnBuf = new_buf;
          r_RgnBufSize = new_buf_size;
     }
     return TRUE;
}

/*
 * R_init_region_with_rect
 *
 * This function initialises a R_Region structure to represent a rectangular
 * region. It is assumed that the region is currently uninitialised.
 *
 * Parameters:
 *        rgn        A pointer to the R_Region to be initialised.
 *        rect       A pointer to an IntXYMinMax structure representing
 *                   the rectangular area requiring an equivalent region
 *                   description.
 * Returns:
 *        TRUE on success, FALSE on failure.
 */
int
R_init_region_with_rect
(
     R_Region       *rgn,
     IntXYMinMax    *rect
```

```
)
{
      R_Int      *rgn_data;

      ASSERT(rect->X.Min <= rect->X.Max);
      ASSERT(rect->Y.Min <= rect->Y.Max);

      rgn->rr_BBox = *rect;
      rgn_data = (R_Int *)malloc(9 * sizeof(R_Int));
      if (rgn_data == NULL)
      {
          return FALSE;
      }
      rgn_data[0] = R_NEXT_IS_Y;
      rgn_data[1] = rect->Y.Min;
      rgn_data[2] = rect->X.Min;
      rgn_data[3] = rect->X.Max + 1;
      rgn_data[4] = R_NEXT_IS_Y;
      rgn_data[5] = rect->Y.Max + 1;
      rgn_data[6] = rect->X.Min;
      rgn_data[7] = rect->X.Max + 1;
      rgn_data[8] = R_EOR;
      rgn->rr_RgnData = rgn_data;
      rgn->rr_RgnDataSize = 9;
      return TRUE;
}

/*
 * R_region_with_region
 *
 * This function initialises a R_Region structure to represent a the region
 * passed as an argument. It is assumed that the region is currently
 * uninitialised.
 *
 * Parameters:
 *        rgn        A pointer to the R_Region to be initialised.
 *        src_rgn    A pointer to an R_Region structure representing
 *                   the region to which this region is to be initialised.
 * Returns:
 *        TRUE on success, FALSE on failure.
 */
int
R_init_region_with_region
(
    R_Region    *rgn,
    R_Region    *src_rgn
)
{
      R_Int      *rgn_data;

      rgn->rr_BBox = src_rgn->rr_BBox;
      rgn_data = (R_Int *)malloc(src_rgn->rr_RgnDataSize * sizeof(R_Int));
      if (rgn_data == NULL)
      {
          return FALSE;
      }
      memcpy
      (
          rgn_data,
          src_rgn->rr_RgnData,
          src_rgn->rr_RgnDataSize * sizeof(R_Int)
      );
      rgn->rr_RgnData = rgn_data;
      rgn->rr_RgnDataSize = src_rgn->rr_RgnDataSize;
      return TRUE;
}

/*
```

```
 * R_region_with_translated_region
 *
 * This function initialises a R_Region structure to represent a the region
 * passed as an argument translated by delta. It is assumed that the region
 * is currently uninitialised.
 *
 * Parameters:
 *        rgn       A pointer to the R_Region to be initialised.
 *        src_rgn   A pointer to an R_Region structure representing
 *                  the region to which this region is to be initialised.
 *        delta     A pointer to a IntXY structure representing the
 *                  translation required.
 * Returns:
 *        TRUE on success, FALSE on failure.
 */
int
R_init_region_with_translated_region
(
    R_Region    *rgn,
    R_Region    *src_rgn,
    IntXY       *delta
)
{
     R_Int      *rgn_data;
     R_Int      *src_data;

     rgn->rr_BBox.X.Min = src_rgn->rr_BBox.X.Min + delta->X;
     rgn->rr_BBox.X.Max = src_rgn->rr_BBox.X.Max + delta->X;
     rgn->rr_BBox.Y.Min = src_rgn->rr_BBox.Y.Min + delta->Y;
     rgn->rr_BBox.Y.Max = src_rgn->rr_BBox.Y.Max + delta->Y;
     rgn_data = (R_Int *)malloc(src_rgn->rr_RgnDataSize * sizeof(R_Int));
     if (rgn_data == NULL)
     {
         return FALSE;
     }
     src_data = src_rgn->rr_RgnData;
     for (int i = 0; i < src_rgn->rr_RgnDataSize; i++)
     {
          if (src_data[i] == R_NEXT_IS_Y)
          {
              rgn_data[i] = src_data[i];
              i++;
              rgn_data[i] = src_data[i] + delta->Y;
              continue;
          }
          else if (src_data[i] == R_EOR)
              rgn_data[i] = src_data[i];
          else
              rgn_data[i] = src_data[i] + delta->X;

     }
     rgn->rr_RgnData = rgn_data;
     rgn->rr_RgnDataSize = src_rgn->rr_RgnDataSize;
     return TRUE;
}

/*
 * R_empty_region
 *
 * Deallocates the region data allocated for a region. Only the
 * data is freed. The R_Region structure itself is not.
 *
 * Parameters:
 *        rgn       The region whose region data is to be deallocated.
 * Returns:
 *        Nothing.
 */
void
```

```
R_empty_region
(
     R_Region   *rgn
)
{
     if (rgn != NULL && rgn->rr_RgnData != NULL)
     {
          free(rgn->rr_RgnData);
          rgn->rr_RgnData = NULL;
     }
}
#ifndef R_USE_NEW_IMP
/*
 * R_union
 *
 * This function inits a R_Region structure to represent the union
 * of it's two arguments.
 *
 * Parameters:
 *          rgn    The R_Region to be initialised.
 *          r1     A R_Region ptr representing the first region.
 *          r2     A R_Region ptr representing the second region.
 * Returns
 *          TRUE on success, FALSE on failure.
 */
int
R_union
(
     R_Region   *rgn,
     R_Region   *r1,
     R_Region   *r2
)
{
     R_Int      *r1_dat;
     R_Int      *r2_dat;
     int        overlap_flags;

     union_tot++;

     if    (!BB_intersect_test(&r1->rr_BBox, &r2->rr_BBox, &overlap_flags))
     {
          /*
           * The bounding boxes don't intersect. This means we can do the
           * union very easily, simply by copying data from the two regions.
           * We malloc a new region data array of size r1->rr_RgnDataSize +
           * r2->rr_RgnDataSize - 1. This is the maximum possible size of
           * resulting region. Not all of this memory will be utilised if
           * the two regions being combined have rows with the same y coordinate
           * (R_NEXT_IS_Y marker is not duplicated).
           */
          rgn->rr_RgnDataSize = r1->rr_RgnDataSize + r2->rr_RgnDataSize - 1;
          rgn->rr_RgnData = (R_Int *)malloc(rgn->rr_RgnDataSize *
                                                    sizeof(R_Int));
          if (rgn->rr_RgnData == NULL)
          {
               return FALSE;
          }
          /*
           * Now, check to see if the regions overlap in y...
           */
          if    (!(overlap_flags & BB_INTERSECT_OVERLAP_Y))
          {
               /*
                * The regions don't overlap in y. We simply copy one region
                * and then another into the array we malloced. We ensure
                * that r1 points to the region with the smallest y coordinate.
                */
               if (r2->rr_BBox.Y.Min < r1->rr_BBox.Y.Min)
```

```
        {
             R_Region   *tmp;
             tmp = r1;
             r1 = r2;
             r2 = tmp;
        }
        memcpy
        (
             rgn->rr_RgnData,
             r1->rr_RgnData,
             (r1->rr_RgnDataSize - 1) * sizeof(R_Int)
        );
        memcpy
        (
             rgn->rr_RgnData + r1->rr_RgnDataSize - 1,
             r2->rr_RgnData,
             r2->rr_RgnDataSize * sizeof(R_Int)
        );
        ASSERT(rgn->rr_RgnData[rgn->rr_RgnDataSize - 1] == R_EOR);
   }
   else
   {
        R_Int      *r1_tmp;
        R_Int      *r2_tmp;
        R_Int      *dest;
        R_Int      min_row;
        int        r1_done;
        int        r1_consumed;
        int        r2_consumed;
        int        num_written;
        /*
         * The bboxes overlap in y but not in x. We simply go row
         * by row through each region and memcpy the individual rows as
         * appropriate. We ensure that r1 points to the region with
         * the smallest x coordinate.
         */
        if (r2->rr_BBox.X.Min < r1->rr_BBox.X.Min)
        {
             R_Region   *tmp;
             tmp = r1;
             r1 = r2;
             r2 = tmp;
        }
        r1_dat = r1->rr_RgnData;
        r2_dat = r2->rr_RgnData;
        dest = rgn->rr_RgnData;
        rgn->rr_RgnDataSize = 0;
        r1_consumed = 0;
        r2_consumed = 0;
        while (*r1_dat != R_EOR && *r2_dat != R_EOR)
        {
             ASSERT(*r1_dat == R_NEXT_IS_Y);
             ASSERT(*r2_dat == R_NEXT_IS_Y);
             min_row = min(r1_dat[1], r2_dat[1]);
             r1_done = FALSE;
             if (r1_dat[1] == min_row)
             {
                  /*
                   * We need to emit r1. We therefore need to find where
                   * the next row (if any) starts. When we do this we
                   * recall that a y value _must be followed by at least
                   * two x values..
                   */
                  r1_tmp = r1_dat + 4;
                  while (*r1_tmp != R_NEXT_IS_Y && *r1_tmp != R_EOR)
                       r1_tmp++;
                  num_written = r1_tmp - r1_dat;
                  memcpy(dest, r1_dat, num_written * sizeof(R_Int));
        dest +=  num_written;
        r1_consumed +=  num_written;
        rgn->rr_RgnDataSize +=  num_written;
        r1_dat = r1_tmp;
        r1_done = TRUE;
    }
    if (r2_dat[1] == min_row)
    {
        /*
         * We need to emit r1. We therefore need to find where
         * the next row (if any) starts. When we do this we
         * recall that a y value _must be followed by at least
         * two x values. If r1's current row has already been
         * emitted for this y value, we do _not_ emit the
         * R_NEXT_IS_Y marker or the y value itself.
         */
        if (r1_done)
        {
            r2_dat += 2;
            r2_tmp = r2_dat + 2;
            r2_consumed += 2;
        }
        else
        {
            r2_tmp = r2_dat + 4;
        }
        while (*r2_tmp != R_NEXT_IS_Y && *r2_tmp != R_EOR)
            r2_tmp++;
        num_written = r2_tmp - r2_dat;
        memcpy(dest, r2_dat, num_written * sizeof(R_Int));
        dest += num_written;
        r2_consumed += num_written;
        rgn->rr_RgnDataSize += num_written;
        r2_dat = r2_tmp;
    }
}
if (*r1_dat != R_EOR)
{
    /*
     * r1 is the last region left standing. We memcpy
     * the remainder of the region (including the
     * R_EOR marker) to the destination.
     */
    ASSERT(r2_consumed == r2->rr_RgnDataSize - 1);
    memcpy
    (
        dest,
        r1_dat,
        (r1->rr_RgnDataSize - r1_consumed) * sizeof(R_Int)
    );
    rgn->rr_RgnDataSize += (r1->rr_RgnDataSize - r1_consumed);
}
else
{
    /*
     * r2 is the last region left standing. We memcpy
     * the remainder of the region (including the
     * R_EOR marker) to the destination.
     */
    ASSERT(r1_consumed == r1->rr_RgnDataSize - 1);
    memcpy
    (
        dest,
        r2_dat,
        (r2->rr_RgnDataSize - r2_consumed) * sizeof(R_Int)
    );
    rgn->rr_RgnDataSize += (r2->rr_RgnDataSize - r2_consumed);
}
```

```
            ASSERT
            (
                 rgn->rr_RgnData[rgn->rr_RgnDataSize - 1] == R_EOR
            );

        }
    }
    else
    {
        R_Int           min_row;
        int             dest_size;
        unsigned char   *rgn_bld_stat;
        R_Int           *rgn_bld_dat;
        int             i;
        int             in_run;
        int             done_r1_in_row;

        union_full++;

        /*
         * The two regions _do_ overlap in x _and y. We therefore have
         * to do a bit more work in calculating the union of the two
         * regions. We use the R_RegionBuilder struct to store state
         * regarding the currently active regions as we progress through
         * the rows of each region. After any rows relevent to a y-coord
         * are added to the region builder, we examine the state of each
         * pixel run in the region builder. If the addition of the row(s)
         * for the y-coord have caused a transition to or from 0, then
         * the pixel run is emitted. However, the first thing we do is
         * ensure the current region builder is empty.
         */
        r1_dat = r1->rr_RgnData;
        r2_dat = r2->rr_RgnData;
        R_CurRB->rrb_Nels = 0;
        dest_size = 0;
        /*
         * We are now ready to loop through the data of both regions.
         * We continue building the new region whilst there is data
         * remaining in either of the two regions.
         */
        while (*r1_dat != R_EOR || *r2_dat != R_EOR)
        {
            ASSERT(*r1_dat == R_NEXT_IS_Y || *r1_dat == R_EOR);
            ASSERT(*r2_dat == R_NEXT_IS_Y || *r2_dat == R_EOR);
            if (*r1_dat == R_EOR)
                min_row = r2_dat[1];
            else if (*r2_dat == R_EOR)
                min_row = r1_dat[1];
            else
                min_row = min(r1_dat[1], r2_dat[1]);
            done_r1_in_row = FALSE;
            if (*r1_dat != R_EOR && r1_dat[1] == min_row)
            {
                /*
                 * The first region is active on this y coord. We add this
                 * row to the current region builder.
                 */
                if (!R_add_row_to_region_builder(&r1_dat, 0x1, TRUE))
                    return FALSE;
                done_r1_in_row = TRUE;
            }
            if (*r2_dat != R_EOR && r2_dat[1] == min_row)
            {
                /*
                 * The first region is active on this y coord. We add this
                 * row to the current region builder.
                 */
                if (!R_add_row_to_region_builder(&r2_dat, 0x2, !done_r1_in_row))
                return FALSE;
        }
        /*
         * Now, we generate the output row for the input rows.
         */
        if (!r_check_rgn_buf_len(dest_size + 2))
        {
             return FALSE;
        }
        r_RgnBuf[dest_size++] = R_NEXT_IS_Y;
        r_RgnBuf[dest_size++] = min_row;
        rgn_bld_stat = R_CurRB->rrb_StateData;
        rgn_bld_dat = R_CurRB->rrb_RgnData;
        in_run = FALSE;
        for (i = R_CurRB->rrb_Nels; i > 0; i--)
        {
             if
             (
                  *rgn_bld_stat > 0
                  &&
                  (
                       (*rgn_bld_stat & RB_CUR_STATE_MASK) == 0
                       ||
                       (*rgn_bld_stat & RB_PREV_STATE_MASK) == 0
                  )
             )
             {
                  /*
                   * We have to emit a run here, if we're not already
                   * in one..
                   */
                  if (!in_run)
                  {
                       if (!r_check_rgn_buf_len(dest_size + 1))
                       {
                            return FALSE;
                       }
                       r_RgnBuf[dest_size++] = *rgn_bld_dat;
                       in_run = TRUE;
                  }
             }
             else
             {
                  if (in_run)
                  {
                       /*
                        * We've come to the end of a run. We output the next
                          element to end it.
                        */
                       if (!r_check_rgn_buf_len(dest_size + 1))
                       {
                            return FALSE;
                       }
                       r_RgnBuf[dest_size++] = *rgn_bld_dat;
                  }
                  in_run = FALSE;
             }
             rgn_bld_stat++;
             rgn_bld_dat++;
        }
        if (r_RgnBuf[dest_size - 2] == R_NEXT_IS_Y)
        {
             /*
              * We didn't output anything for these input rows. Rewind..
              */
             dest_size -= 2;
        }
}
```

```
            /*
             * We've completed constructing the data for the region. We
             * make a copy the constructed data from the permanent buffer to
             * an exactly fitting buffer.
             */
            rgn->rr_RgnData = (R_Int *)malloc(++dest_size * sizeof(R_Int));
            if (rgn->rr_RgnData == NULL)
            {
                return FALSE;
            }
            memcpy(rgn->rr_RgnData, r_RgnBuf, (dest_size - 1) * sizeof(R_Int));
            rgn->rr_RgnData[dest_size - 1] = R_EOR;
            rgn->rr_RgnDataSize = dest_size;
            ASSERT(rgn->rr_RgnDataSize >= 9);
        }
        /*
         * We now do a bounding box union of the two component bboxes and place
         * the result in the new region.
         */
        BB_union(&r1->rr_BBox, &r2->rr_BBox, &rgn->rr_BBox);
        /*
         * Done! We can get out..
         */
        return TRUE;
}

/*
 * R_union_equals
 *
 * This function basically implements a r1 union= r2 type operation. Ie
 * r1 union r2 is calculated and the result returned in r1.
 *
 * Parameters:
 *      r1          A pointer to an R_Region. This represents
 *                  the first half of the union, and is also used to return
 *                  the eventual result.
 *      r2          A pointer to an R_Region. This represents the second
 *                  half of the union.
 * Returns:
 *      TRUE on success, FALSE on failure.
 */
int
R_union_equals
(
        R_Region    *r1,
        R_Region    *r2
)
{
        R_Region    new_rgn;
        if (r1->rr_RgnData == NULL)
            return R_init_region_with_region(r1, r2);
        if (!R_union(&new_rgn, r1, r2))
            return FALSE;
        R_empty_region(r1);
        *r1 = new_rgn;
        return TRUE;
}

/*
 * R_intersection
 *
 * This function inits a R_Region structure to represent the intersection
 * of it's two arguments.
 *
 * Parameters:
 *      rgn         A R_Region ptr to the R_Region structure to be initialised.
 *      r1          A R_Region ptr representing the first region.
 *      r2          A R_Region ptr representing the second region.
 * Returns
 *        TRUE on success, FALSE on failure.
 */
int
R_intersection
(
     R_Region   *rgn,
     R_Region   *r1,
     R_Region   *r2
)
{
     R_Int      *r1_dat;
     R_Int      *r2_dat;
     int        overlap_flags;


     int_tot++;

     rgn->rr_RgnData = NULL;
     if (!BB_intersect_test(&r1->rr_BBox, &r2->rr_BBox, &overlap_flags))
     {
          /*
           * The bounding boxes don't intersect. This means that the regions
           * don't intersect. Therefore, we simply set rgn->rr_RgnData to NULL
           * (signifying an empty region) and get out..
           */
          return TRUE;
     }
     R_Int                min_row;
     int                  dest_size;
     unsigned char        *rgn_bld_stat;
     R_Int                *rgn_bld_dat;
     int                  i;
     int                  in_run;
     int                  done_r1_in_row;
     IntXYMinMax          new_bbox;

     int_full++;

     /*
      * The two regions _do_ overlap in x _and y. We therefore have
      * to do a bit more work in calculating the intersection of the two
      * regions. We use the R_RegionBuilder struct to store state
      * regarding the currently active regions as we progress through
      * the rows of each region. After any rows relevent to a y-coord
      * are added to the region builder, we examine the state of each
      * pixel run in the region builder. If the addition of the row(s)
      * for the y-coord have caused a transition to or from 0x3, then
      * the pixel run is emitted.
      */
     /*
      * Initialise the new_bbox structure for determining the new bounding box.
      */
     new_bbox.X.Min = R_INT_MAX_VALUE;
     new_bbox.Y.Min = R_INT_MAX_VALUE;
     new_bbox.X.Max = R_INT_MIN_VALUE;
     new_bbox.Y.Max = R_INT_MIN_VALUE;

     /*
      * The next thing we do is ensure the current region builder is empty,
      * and set up pointers into the region data of the two regions.
      */
     r1_dat = r1->rr_RgnData;
     r2_dat = r2->rr_RgnData;
     R_CurRB->rrb_Nels = 0;
     dest_size = 0;
     /*
      * We are now ready to loop through the data from both regions. Notice
 * that we only keep looping whilst _both_ regions have some data left
 * to give. As soon as either of the region's data has been exhausted,
 * then we stop as the intersection region has already been calculated
 * and is sitting in the rgn_buf.
 */
while (*r1_dat != R_EOR && *r2_dat != R_EOR)
{
    ASSERT(*r1_dat == R_NEXT_IS_Y || *r1_dat == R_EOR);
    ASSERT(*r2_dat == R_NEXT_IS_Y || *r2_dat == R_EOR);
    if (*r1_dat == R_EOR)
        min_row = r2_dat[1];
    else if (*r2_dat == R_EOR)
        min_row = r1_dat[1];
    else
        min_row = min(r1_dat[1], r2_dat[1]);
    done_r1_in_row = FALSE;
    if (*r1_dat != R_EOR && r1_dat[1] == min_row)
    {
        /*
         * The first region is active on this y coord. We add this
         * row to the current region builder.
         */
        if (!R_add_row_to_region_builder(&r1_dat, 0x1, TRUE))
            return FALSE;
        done_r1_in_row = TRUE;
    }
    if (*r2_dat != R_EOR && r2_dat[1] == min_row)
    {
        /*
         * The first region is active on this y coord. We add this
         * row to the current region builder.
         */
        if (!R_add_row_to_region_builder(&r2_dat, 0x2, !done_r1_in_row))
            return FALSE;
    }
    /*
     * Now, we generate the output row for the input rows.
     */
    if (!r_check_rgn_buf_len(dest_size + 2))
    {
       return FALSE;
    }
    r_RgnBuf[dest_size++] = R_NEXT_IS_Y;
    r_RgnBuf[dest_size++] = min_row;
    rgn_bld_stat = R_CurRB->rrb_StateData;
    rgn_bld_dat = R_CurRB->rrb_RgnData;
    in_run = FALSE;
    for (i = R_CurRB->rrb_Nels; i > 0; i--)
    {
        if
        (
            *rgn_bld_stat != (3 | (3 << RB_STATE_SIZE))
            &&
            (
                (*rgn_bld_stat & RB_PREV_STATE_MASK) == 3
                ||
                (*rgn_bld_stat & RB_CUR_STATE_MASK) == (3 << RB_STATE_SIZE))
        )
        {
            /*
             * We have to emit a run here, if we're not already
             * in one..
             */
            if (!in_run)
            {
                if (!r_check_rgn_buf_len(dest_size + 1))
                {
                    return FALSE;
                    }
                    r_RgnBuf[dest_size++] = *rgn_bld_dat;
                    in_run = TRUE;
                    new_bbox.X.Min = min(new_bbox.X.Min, *rgn_bld_dat);
               }
          }
          else
          {
               if (in_run)
               {
                    /*
                     * We've come to the end of a run. We output the next element
                       to end it.
                     */
                    if (!r_check_rgn_buf_len(dest_size + 1))
                    {
                        return FALSE;
                    }
                    r_RgnBuf[dest_size++] = *rgn_bld_dat;
                    new_bbox.X.Max = max(new_bbox.X.Max, *rgn_bld_dat);
               }
               in_run = FALSE;
          }
          rgn_bld_stat++;
          rgn_bld_dat++;
     }
     if (r_RgnBuf[dest_size - 2] == R_NEXT_IS_Y)
     {
          /*
           * We didn't output anything for these input rows. Rewind..
           */
          dest_size -= 2;
     }
     else
     {
          if (min_row < new_bbox.Y.Min)
              new_bbox.Y.Min = min_row;
          else if (min_row > new_bbox.Y.Max)
              new_bbox.Y.Max = min_row;
     }
}
/*
 * We've completed constructing the data for the region. Firstly
 * we check to see if we've emitted anything at all. If we have
 * then dest_size must be > 0. If it isn't we simply free the
 * region we created and get out, as the regions don't really
 * intersect, in spite of their intersecting bounding boxes.
 */
if (dest_size == 0)
{
     return TRUE;
}
/*
 * We make a copy the constructed data from the permanent buffer to
 * an exactly fitting buffer.
 */
rgn->rr_RgnData = (R_Int *)malloc(++dest_size * sizeof(R_Int));
if (rgn->rr_RgnData == NULL)
{
     return FALSE;
}
memcpy(rgn->rr_RgnData, r_RgnBuf, (dest_size - 1) * sizeof(R_Int));
rgn->rr_RgnData[dest_size - 1] = R_EOR;
rgn->rr_RgnDataSize = dest_size;
ASSERT(rgn->rr_RgnDataSize >= 9);
/*
 * Now, copy across the bounding box.. Before we do this, we subtract
 * 1 from X.Max and Y.Max because of the region format.
```

```
     */
    new_bbox.X.Max--;
    new_bbox.Y.Max--;
    rgn->rr_BBox = new_bbox;
    /*
     * Done! We can get out..
     */
    return TRUE;
}

/*
 * R_difference
 *
 * This function inits a R_Region structure to represent the difference of
 * it's two arguments. It essentially calculates r1 - r2
 *
 * Parameters:
 *          rgn         A R_Region ptr representing the R_Region to be inited.
 *          r1          A R_Region ptr representing the first region.
 *          r2          A R_Region ptr representing the second region.
 * Returns
 *          TRUE on success, FALSE on failure.
 */
int
R_difference
(
    R_Region    *rgn,
    R_Region    *r1,
    R_Region    *r2
)
{
    R_Int       *r1_dat;
    R_Int       *r2_dat;
    int         overlap_flags;


    diff_tot++;

    rgn->rr_RgnData = NULL;
    if (!BB_intersect_test(&r1->rr_BBox, &r2->rr_BBox, &overlap_flags))
    {
        /*
         * The bounding boxes don't intersect. This means that r1 - r2
         * simply equals r1. We make a copy of the relevant bits and get out..
         */
        rgn->rr_BBox = r1->rr_BBox;
        rgn->rr_RgnDataSize = r1->rr_RgnDataSize;
        rgn->rr_RgnData = (R_Int *)malloc(r1->rr_RgnDataSize * sizeof(R_Int));
        if (rgn->rr_RgnData == NULL)
        {
            return FALSE;
        }
        memcpy
        (
            rgn->rr_RgnData,
            r1->rr_RgnData,
            r1->rr_RgnDataSize * sizeof(R_Int)
        );
        return TRUE;
    }
    R_Int           min_row;
    int             dest_size;
    unsigned char   *rgn_bld_stat;
    R_Int           *rgn_bld_dat;
    int             i;
    int             in_run;
    int             done_r1_in_row;
    unsigned char   m_high;
unsigned char    m_low;
IntXYMinMax      new_bbox;

diff_full++;

/*
 * The two regions _do_ overlap in x _and y. We therefore have
 * to do a bit more work in calculating the difference of the two
 * regions. We use the R_RegionBuilder struct to store state
 * regarding the currently active regions as we progress through
 * the rows of each region. After any rows relevent to a y-coord
 * are added to the region builder, we examine the state of each
 * pixel run in the region builder. If the addition of the row(s)
 * for the y-coord have caused the following transitions -
 *              r1      -> 0
 *              0       -> r2
 *              r1 + r2 -> r2
 *              r2      -> r1 + r2
 * ..then the relevent runs are emitted. Firstly, though,
 * we ensure the current region builder is empty,
 * and set up pointers into the region data of the two regions.
 */
r1_dat = r1->rr_RgnData;
r2_dat = r2->rr_RgnData;
R_CurRB->rrb_Nels = 0;
dest_size = 0;

/*
 * Initialise the new_bbox structure for determining the new bounding box.
 */
new_bbox.X.Min = 32767;
new_bbox.Y.Min = 32767;
new_bbox.X.Max = -32768;
new_bbox.Y.Max = -32768;

/*
 * We are now ready to loop through the data from both regions. Notice
 * that we only keep looping whilst r1 has data outstanding. When
 * r1's data is consumed, then any transitions made by r2 are
 * irrelevant.
 */
while (*r1_dat != R_EOR)
{
      ASSERT(*r1_dat == R_NEXT_IS_Y || *r1_dat == R_EOR);
      ASSERT(*r2_dat == R_NEXT_IS_Y || *r2_dat == R_EOR);
      if (*r1_dat == R_EOR)
          min_row = r2_dat[1];
      else if (*r2_dat == R_EOR)
          min_row = r1_dat[1];
      else
          min_row = min(r1_dat[1], r2_dat[1]);
      done_r1_in_row = FALSE;
      if (*r1_dat != R_EOR && r1_dat[1] == min_row)
      {
            /*
             * The first region is active on this y coord. We add this
             * row to the current region builder.
             */
            if (!R_add_row_to_region_builder(&r1_dat, 0x1, TRUE))
                return FALSE;
            done_r1_in_row = TRUE;
      }
      if (*r2_dat != R_EOR && r2_dat[1] == min_row)
      {
            /*
             * The first region is active on this y coord. We add this
             * row to the current region builder.
             */
          if (!R_add_row_to_region_builder(&r2_dat, 0x2, !done_r1_in_row))
              return FALSE;
     }
     /*
      * Now, we generate the output row for the input rows.
      */
     if   (!r_check_rgn_buf_len(dest_size + 2))
     {
          return FALSE;
     }
     r_RgnBuf[dest_size++] = R_NEXT_IS_Y;
     r_RgnBuf[dest_size++] = min_row;
     rgn_bld_stat = R_CurRB->rrb_StateData;
     rgn_bld_dat = R_CurRB->rrb_RgnData;
     in_run = FALSE;
     for (i = R_CurRB->rrb_Nels; i > 0; i--)
     {
          m_high = (*rgn_bld_stat & RB_CUR_STATE_MASK) >> RB_STATE_SIZE;
          m_low = *rgn_bld_stat & RB_PREV_STATE_MASK;
          if
          (
               (
                    (m_low != 1 && m_high == 1)
                    ||
                    (m_low == 1 && m_high != 1)
               )
          )
          {
               /*
                * We have to emit a run here, if we're not already
                * in one..
                */
               if (!in_run)
               {
                    if (!r_check_rgn_buf_len(dest_size + 1))
                    {
                         return FALSE;
                    }
                    r_RgnBuf[dest_size++] = *rgn_bld_dat;
                    in_run = TRUE;
                    new_bbox.X.Min = min(new_bbox.X.Min, *rgn_bld_dat);
               }
          }
          else
          {
               if (in_run)
               {
                    /*
                     * We've come to the end of a run. We output the next element
                       to end it.
                     */
                    if (!r_check_rgn_buf_len(dest_size + 1))
                    {
                        return FALSE;
                    }
                    r_RgnBuf[dest_size++] = *rgn_bld_dat;
                    new_bbox.X.Max = max(new_bbox.X.Max, *rgn_bld_dat);
               }
               in_run = FALSE;
          }
          rgn_bld_stat++;
          rgn_bld_dat++;
     }
     if   (r_RgnBuf[dest_size - 2] == R_NEXT_IS_Y)
     {
          /*
           * We didn't output anything for these input rows. Rewind..
           */
```

```
                    dest_size -= 2;
                }
                else
                {
                    if (min_row < new_bbox.Y.Min)
                        new_bbox.Y.Min = min_row;
                    else if (min_row > new_bbox.Y.Max)
                        new_bbox.Y.Max = min_row;
                }
            }
            /*
             * We've completed constructing the data for the region. Firstly
             * we check to see if we've emitted anything at all. If we have
             * then dest_size must be > 0. If it isn't we simply free the
             * region we created and get out, as r2 - r1 must be empty.
             */
            if (dest_size == 0)
            {
                return TRUE;
            }
            /*
             * We make a copy the constructed data from the permanent buffer to
             * an exactly fitting buffer.
             */
            rgn->rr_RgnData = (R_Int *)malloc(++dest_size * sizeof(R_Int));
            if (rgn->rr_RgnData == NULL)
            {
                return FALSE;
            }
            memcpy(rgn->rr_RgnData, r_RgnBuf, (dest_size - 1) * sizeof(R_Int));
            rgn->rr_RgnData[dest_size - 1] = R_EOR;
            rgn->rr_RgnDataSize = dest_size;
            ASSERT(rgn->rr_RgnDataSize >= 9);
            /*
             * Now, copy across the bounding box..
             */
            rgn->rr_BBox = new_bbox;
            /*
             * Done! We can get out..
             */
            return TRUE;
        }
        #endif /* R_USE_NEW_IMP */
        /*
         * r_grow_free_list
         *
         * This function mallocs and adds R_FREE_LIST_GROWTH_SIZE new elements
         * to the front of the region growth free list.
         *
         * Parameters:
         *          None.
         * Returns:
         *          TRUE on success, FALSE on failure.
         */
        int
        r_grow_free_list()
        {
            R_RgnGrowItem   *rgi;
            int             i;
            /*
             * First, malloc the memory..
             */
            rgi =   (R_RgnGrowItem *)malloc
                    (
                        R_FREE_LIST_GROWTH_SIZE * sizeof(R_RgnGrowItem)
                    );
            if (rgi == NULL)
                return FALSE;
        /*
         * Now make the whole block of memory into a list..
         */
        for (i = 0; i < R_FREE_LIST_GROWTH_SIZE - 1; i++)
            rgi[i].rrgi_Next = &rgi[i + 1];
        /*
         * Now, add it to the front of the free list..
         */
        rgi[R_FREE_LIST_GROWTH_SIZE - 1].rrgi_Next = r_free_list;
        r_free_list = rgi;
        return TRUE;
}

/*
 * R_add_row_to_region_growth_list
 *
 * This function adds a row from a R_Region to a linked list comprised of
 * R_RgnGrowItem structures. "Adding" implies that the linked list is
 * modified such that the state and coordinate information present
 * in the list is updated to take into account the new row just added.
 * Adding a row has the following properties:
 *         * If a pixel run in the row does not exist in the list before,
 *           it is added and it's current state is tagged with the region
 *           to which the row belongs. The previous state is set to 0,
 *           indicating that it did not exist before.
 *         * If a pixel run in the row did exist before, but it's present state
 *           indicates that it came from the other region then the run
 *           is retained but it's state is modified to indicate that
 *           both regions are active at this point.
 *         * If a pixel run in the row did exist before, and it's present
 *           state indicates that the current region then the region is
 *           removed and it's state is modified to indicate that the run is
 *           now empty.
 *         * If a pixel run in the row did exist before, and it's present state
 *           indicates that both regions are currently active then the run
 *           is retained, but its state is modified to indicate that only the
 *           other region is active in this run.
 *
 * Parameters:
 *         row_ptr    A R_Int ** pointer to the row in the region. Used
 *                    to return the updates row pointer.
 *         rgn_mask   A mask for the region the row comes from. Must
 *                    be either 1 or 2.
 *         first      Whether this is the first region to be processed
 *                    on the current scanline.
 * Returns:
 *         TRUE on success, FALSE on failure.
 */
int
R_add_row_to_region_growth_list
(
        R_Int       **row_ptr,
        int         rgn_mask,
        int         first
)
{
        R_Int              *row;
        R_RgnGrowItem      *rgi;
        unsigned char      rb_prev_run_state;
        int                row_on;

        row = *row_ptr;
        /*
         * Skip over the row's y value at the beginning.
         */
        ASSERT(*row == R_NEXT_IS_Y);
        row += 2;
        ASSERT(*row != R_NEXT_IS_Y && *row != R_EOR);
        If    (r_growth_list == NULL)
        {
              R_RgnGrowItem    **ptr_next_ptr;
              /*
               * The growth list is currently empty. Therefore, we simply convert
               * the input row to the region growth list format..
               */
              row_on = TRUE;
              ptr_next_ptr = &r_growth_list;
              while (R_NOT_END_OF_ROW(*row))
              {
                    if (r_free_list == NULL)
                    {
                          if (!r_grow_free_list())
                              return FALSE;
                    }
                    rgi = r_free_list;
                    *ptr_next_ptr = rgi;
                    ptr_next_ptr = &rgi->rrgi_Next;
                    r_free_list = *ptr_next_ptr;
                    rgi->rrgi_RgnData = *row;
                    if    (row_on)
                          rgi->rrgi_StateData = (rgn_mask << RB_STATE_SIZE);
                    else
                          rgi->rrgi_StateData = 0;
                    row_on = !row_on;
                    row++;
              }
              *row_ptr = row;
              *ptr_next_ptr = NULL;
              return TRUE;
        }

        R_RgnGrowItem    fake_item;
        R_RgnGrowItem    *prev_rgi;

        /*
         * "fake_item" is used as the head of the list. This is so that we _always_ have
         * a valid pointer to the previous item in the list. Only the next pointer and
         * state data are initialised, as these are they only elements which will be
         * referenced.
         */
        fake_item.rrgi_StateData = 0;
        fake_item.rrgi_Next = r_growth_list;
        prev_rgi = &fake_item;

        if    (first)
        {
              /*
               * If this is the first row to be added on this particular scanline,
               * then we have to update the existing contents of those elements
               * at the beginning of the growth list which precede (in coords) the
               * first element of the row. "Updating" involves updating the
               * previous state of each element to match the current state. This is
               * because none of the elements were effected by the addition of the
               * new row.
               */
              rgi = r_growth_list;
              while (rgi != NULL && *row > rgi->rrgi_RgnData)
              {
#ifndef RB_USE_LOOKUP
                    rgi->rrgi_StateData = (rgi->rrgi_StateData & RB_CUR_STATE_MASK) |
                                          (rgi->rrgi_StateData >> RB_STATE_SIZE);
#else
                    rgi->rrgi_StateData = r_shift_and_dup[rgi->rrgi_StateData];
#endif
                    prev_rgi = rgi;
```

```
            rgi = rgi->rrgi_Next;
        }
    }
    else
    {
        /*
         * This is the second row to be added on this particular scanline.
         * Therefore, we don't need to update the state of the elements
         * preceding (in coords) the first run of the row to be added, as
         * they have already been updated by the first row to be added on
         * this scanline. We simply skip over the unaffected elements..
         */
        rgi = r_growth_list;
        while (rgi != NULL && *row > rgi->rrgi_RgnData)
        {
            prev_rgi = rgi;
            rgi = rgi->rrgi_Next;
        }
    }
    if (rgi == NULL)
    {
        /*
         * We've already exhausted the current growth list. Set the start
         * of the next pixel run to be the max. possible and set the state
         * to be 0.
         */
        rb_prev_run_state = 0;
    }
    else
    {
        /*
         * We are still within the current growth list bounds. Set up
         * the run info appropriately.
         */
        if (rgi == r_growth_list)
            rb_prev_run_state = 0;
        else
            rb_prev_run_state = prev_rgi->rrgi_StateData;
    }

    /*
     * We can now start merging the elements of the row with the remaining
     * elements of the growth list.
     */
    row_on = TRUE;
    while (R_NOT_END_OF_ROW(*row))
    {
        if   (rgi == NULL || *row < rgi->rrgi_RgnData)
        {
            /*
             * This is the only situation in which we actually have to
             * create a new list element. First, we check that we
             * actually have an element in the free list that we
             * can use in the growth list..
             */
            if (r_free_list == NULL)
            {
                if (!r_grow_free_list())
                    return FALSE;
            }
            prev_rgi->rrgi_Next = r_free_list;
            prev_rgi = r_free_list;
            r_free_list = r_free_list->rrgi_Next;
            prev_rgi->rrgi_Next = rgi;
            /*
             * Now, fill in the data..
             */
            prev_rgi->rrgi_RgnData = *row;
```

```
                if   (first)
                {
                     /*
                      * We are processing the first region. Therefore, we
                      * copy the current state of the run to the lowest
                      * RB_STATE_SIZE bits.
                      */
#ifndef RB_USE_LOOKUP
                    prev_rgi->rrgi_StateData = (rb_prev_run_state & RB_CUR_STATE_MASK) |
                                               (rb_prev_run_state >> RB_STATE_SIZE);
#else
                    prev_rgi->rrgi_StateData = r_shift_and_dup[rb_prev_run_state];
#endif
                }
                else
                {
                     /*
                      * We are processing the second region. Therefore, the state data
                      * has already been copied to the previous state area so we
                      * just copy the state.
                      */
                     prev_rgi->rrgi_StateData = rb_prev_run_state;
                }
                /*
                 * Now, if the row for the current region is active at this transition,
                 * we xor the region mask with the current contents of the new list
                 * item.This gives the desired behaviour of making that region active
                 * if it is not there already, but turns it off if it is...
                 */
                if (row_on)
                    prev_rgi->rrgi_StateData ^=  (rgn_mask << RB_STATE_SIZE);
                /*
                 * We now move onto the next row element.
                 */
                row++;
                row_on = !row_on;
                continue;
            }
            /*
             * If the current row transition point is equal in x position to the current
             * list item's transition point, we advance the row counter to
             * the next position.
             */
            if (*row == rgi->rrgi_RgnData)
            {
                row++;
                row_on = !row_on;
            }
            /*
             * We update the current list item to deal with the affects of the
             * current row run..
             */
            rb_prev_run_state = rgi->rrgi_StateData;
            if (first)
            {
#ifndef RB_USE_LOOKUP
                rgi->rrgi_StateData =  (rb_prev_run_state & RB_CUR_STATE_MASK) |
                                       (rb_prev_run_state >> RB_STATE_SIZE);
#else
                rgi->rrgi_StateData = r_shift_and_dup[rb_prev_run_state];
#endif
            }
            if (!row_on)
                rgi->rrgi_StateData ^=  (rgn_mask << RB_STATE_SIZE);
            /*
             * We now move onto the next element in the list..
             */
            prev_rgi = rgi;
```

09627395 . 061099

```
          rgi = rgi->rrgi_Next;
     }
     /*
      * Now, simply update the remainder of the elements in the list..
      */
     if   (first)
     {
          while (rgi != NULL)
          {
#ifndef RB_USE_LOOKUP
               rgi->rrgi_StateData =  (rgi->rrgi_StateData & RB_CUR_STATE_MASK) |
                                      (rgi->rrgi_StateData >> RB_STATE_SIZE);
#else
               rgi->rrgi_StateData = r_shift_and_dup[rgi->rrgi_StateData];
#endif
               rgi = rgi->rrgi_Next;
          }
     }
     /*
      * Now copy "fake_item"'s next pointer to r_growth_list, as it will have
      * changed if something was added to the head of the list..
      */
     r_growth_list = fake_item.rrgi_Next;
     /*
      * Update the return pointer to the region data..
      */
     *row_ptr = row;
     /*
      * Everything should now be OK..
      */
     return TRUE;
}

#ifdef R_USE_NEW_IMP

/*
 * r_union_test_table
 *
 * A 16-int lookup table which when provided with an unsigned char
 * of the following form xxyy, will provide evaluate the key
 * state transition test of the union construction loop.
 * Note that R_STATE_SIZE _must_ be 2 for this lookup table to
 * work.
 */
int r_union_test_table[16] = {
                                   0, 1, 1, 1,
                                   1, 0, 0, 0,
                                   1, 0, 0, 0,
                                   1, 0, 0, 0
                              };

/*
 * R_union
 *
 * This function inits a R_Region structure to represent the union
 * of it's two arguments.
 *
 * Parameters:
 *        rgn        The R_Region to be initialised.
 *        r1         A R_Region ptr representing the first region.
 *        r2         A R_Region ptr representing the second region.
 * Returns
 *        TRUE on success, FALSE on failure.
 */
int
R_union
(
    R_Region    *rgn,
```

09837569 . 031599

```
        R_Region    *r1,
        R_Region    *r2
)
{
        R_Int       *r1_dat;
        R_Int       *r2_dat;
        int         overlap_flags;

        union_tot++;

        if (!BB_intersect_test(&r1->rr_BBox, &r2->rr_BBox, &overlap_flags))
        {
            /*
             * The bounding boxes don't intersect. This means we can do the
             * union very easily, simply by copying data from the two regions.
             * We malloc a new region data array of size r1->rr_RgnDataSize +
             * r2->rr_RgnDataSize - 1. This is the maximum possible size of
             * resulting region. Not all of this memory will be utilised if
             * the two regions being combined have rows with the same y coordinate
             * (R_NEXT_IS_Y marker is not duplicated).
             */
            rgn->rr_RgnDataSize = r1->rr_RgnDataSize + r2->rr_RgnDataSize - 1;
            rgn->rr_RgnData = (R_Int *)malloc(rgn->rr_RgnDataSize *
                                                      sizeof(R_Int));
            if (rgn->rr_RgnData == NULL)
            {
                return FALSE;
            }
            /*
             * Now, check to see if the regions overlap in y...
             */
            if (!(overlap_flags & BB_INTERSECT_OVERLAP_Y))
            {
                /*
                 * The regions don't overlap in y. We simply copy one region
                 * and then another into the array we malloced. We ensure
                 * that r1 points to the region with the smallest y coordinate.
                 */
                if (r2->rr_BBox.Y.Min < r1->rr_BBox.Y.Min)
                {
                    R_Region    *tmp;
                    tmp = r1;
                    r1 = r2;
                    r2 = tmp;
                }
                memcpy
                (
                    rgn->rr_RgnData,
                    r1->rr_RgnData,
                    (r1->rr_RgnDataSize - 1) * sizeof(R_Int)
                );
                memcpy
                (
                    rgn->rr_RgnData + r1->rr_RgnDataSize - 1,
                    r2->rr_RgnData,
                    r2->rr_RgnDataSize * sizeof(R_Int)
                );
                ASSERT(rgn->rr_RgnData[rgn->rr_RgnDataSize - 1] == R_EOR);
            }
            else
            {
                R_Int       *r1_tmp;
                R_Int       *r2_tmp;
                R_Int       *dest;
                R_Int       min_row;
                int         r1_done;
                int         r1_consumed;
                int         r2_consumed;
int         num_written;
/*
 * The bboxes overlap in y but not in x. We simply go row
 * by row through each region and memcpy the individual rows as
 * appropriate. We ensure that r1 points to the region with
 * the smallest x coordinate.
 */
if (r2->rr_BBox.X.Min < r1->rr_BBox.X.Min)
{
     R_Region   *tmp;
     tmp = r1;
     r1 = r2;
     r2 = tmp;
}
r1_dat = r1->rr_RgnData;
r2_dat = r2->rr_RgnData;
dest = rgn->rr_RgnData;
rgn->rr_RgnDataSize = 0;
r1_consumed = 0;
r2_consumed = 0;
while (*r1_dat != R_EOR && *r2_dat != R_EOR)
{
     ASSERT(*r1_dat == R_NEXT_IS_Y);
     ASSERT(*r2_dat == R_NEXT_IS_Y);
     min_row = min(r1_dat[1], r2_dat[1]);
     r1_done = FALSE;
     if (r1_dat[1] == min_row)
     {
          /*
           * We need to emit r1. We therefore need to find where
           * the next row (if any) starts. When we do this we
           * recall that a y value _must be followed by at least
           * two x values..
           */
          r1_tmp = r1_dat + 4;
          while (*r1_tmp != R_NEXT_IS_Y && *r1_tmp != R_EOR)
               r1_tmp++;
          num_written = r1_tmp - r1_dat;
          memcpy(dest, r1_dat, num_written * sizeof(R_Int));
          dest +=  num_written;
          r1_consumed +=  num_written;
          rgn->rr_RgnDataSize +=  num_written;
          r1_dat = r1_tmp;
          r1_done = TRUE;
     }
     if (r2_dat[1] == min_row)
     {
          /*
           * We need to emit r1. We therefore need to find where
           * the next row (if any) starts. When we do this we
           * recall that a y value _must be followed by at least
           * two x values. If r1's current row has already been
           * emitted for this y value, we do _not_ emit the
           * R_NEXT_IS_Y marker or the y value itself.
           */
          if (r1_done)
          {
               r2_dat += 2;
               r2_tmp = r2_dat + 2;
               r2_consumed += 2;
          }
          else
          {
               r2_tmp = r2_dat + 4;
          }
          while (*r2_tmp != R_NEXT_IS_Y && *r2_tmp != R_EOR)
               r2_tmp++;
          num_written = r2_tmp - r2_dat;
```

```
                    memcpy(dest, r2_dat, num_written * sizeof(R_Int));
                    dest += num_written;
                    r2_consumed += num_written;
                    rgn->rr_RgnDataSize += num_written;
                    r2_dat = r2_tmp;
               }
          }
          if (*r1_dat != R_EOR)
          {
               /*
                * r1 is the last region left standing. We memcpy
                * the remainder of the region (including the
                * R_EOR marker) to the destination.
                */
               ASSERT(r2_consumed == r2->rr_RgnDataSize - 1);
               memcpy
               (
                    dest,
                    r1_dat,
                    (r1->rr_RgnDataSize - r1_consumed) * sizeof(R_Int)
               );
               rgn->rr_RgnDataSize += (r1->rr_RgnDataSize - r1_consumed);
          }
          else
          {
               /*
                * r2 is the last region left standing. We memcpy
                * the remainder of the region (including the
                * R_EOR marker) to the destination.
                */
               ASSERT(r1_consumed == r1->rr_RgnDataSize - 1);
               memcpy
               (
                    dest,
                    r2_dat,
                    (r2->rr_RgnDataSize - r2_consumed) * sizeof(R_Int)
               );
               rgn->rr_RgnDataSize += (r2->rr_RgnDataSize - r2_consumed);
          }
          ASSERT
          (
               rgn->rr_RgnData[rgn->rr_RgnDataSize - 1] == R_EOR
          );

     }
}
else
{
     R_Int          min_row;
     int            dest_size;
     R_RgnGrowItem  *rgi;
     R_RgnGrowItem  *rgi_tail;
     int            in_run;
     int            done_r1_in_row;

     union_full++;

     /*
      * The two regions _do_ overlap in x _and_ y. We therefore have
      * to do a bit more work in calculating the union of the two
      * regions. We use the a list of R_RgnGrowItem structs to store state
      * regarding the currently active regions as we progress through
      * the rows of each region. After any rows relevent to a y-coord
      * are added to the list, we examine the state of each
      * pixel run in the list. If the addition of the row(s)
      * for the y-coord have caused a transition to or from 0, then
      * the pixel run is emitted.
      */
```

```
            r1_dat = r1->rr_RgnData;
            r2_dat = r2->rr_RgnData;
            dest_size = 0;
            /*
             * We are now ready to loop through the data of both regions.
             * We continue building the new region whilst there is data
             * remaining in either of the two regions.
             */
            while (*r1_dat != R_EOR || *r2_dat != R_EOR)
            {
                  ASSERT(*r1_dat == R_NEXT_IS_Y || *r1_dat == R_EOR);
                  ASSERT(*r2_dat == R_NEXT_IS_Y || *r2_dat == R_EOR);
                  if (*r1_dat == R_EOR)
                      min_row = r2_dat[1];
                  else if (*r2_dat == R_EOR)
                      min_row = r1_dat[1];
                  else
                      min_row = min(r1_dat[1], r2_dat[1]);
                  done_r1_in_row = FALSE;
                  if (*r1_dat != R_EOR && r1_dat[1] == min_row)
                  {
                        /*
                         * The first region is active on this y coord. We add this
                         * row to the current region builder.
                         */
                        if (!R_add_row_to_region_growth_list(&r1_dat, 0x1, TRUE))
                            return FALSE;
                        done_r1_in_row = TRUE;
                  }
                  if (*r2_dat != R_EOR && r2_dat[1] == min_row)
                  {
                        /*
                         * The first region is active on this y coord. We add this
                         * row to the current region builder.
                         */
                        if (!R_add_row_to_region_growth_list(&r2_dat, 0x2,
                              !done_r1_in_row))
                              return FALSE;
                  }
                  /*
                   * Now, we generate the output row for the input rows.
                   */
                  if (!r_check_rgn_buf_len(dest_size + 2))
                  {
                        return FALSE;
                  }
                  r_RgnBuf[dest_size++] = R_NEXT_IS_Y;
                  r_RgnBuf[dest_size++] = min_row;
                  in_run = FALSE;
#ifndef R_NEW_IMP_CONSTRUCTION_LOOP
                  for (rgi = r_growth_list; rgi != NULL; rgi = rgi->rrgi_Next)
                  {
#if 0
                        if
                        (
                              rgi->rrgi_StateData > 0
                              &&
                              (
                                    (rgi->rrgi_StateData & RB_CUR_STATE_MASK) == 0
                                    ||
                                    (rgi->rrgi_StateData & RB_PREV_STATE_MASK) == 0
                              )
                        )
#else
                        if (r_union_test_table[rgi->rrgi_StateData])
#endif
                        {
                              /*
```

09/05/96 · 0983750

```
                         * We have to emit a run here, if we're not already
                         * in one..
                         */
                        if (!in_run)
                        {
                             if (!r_check_rgn_buf_len(dest_size + 1))
                             {
                                return FALSE;
                             }
                             r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
                             in_run = TRUE;
                        }
                   }
                   else
                   {
                        if (in_run)
                        {
                             /*
                              * We've come to the end of a run. We output the next
                                element to end it.
                              */
                             if (!r_check_rgn_buf_len(dest_size + 1))
                             {
                                  return FALSE;
                             }
                             r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
                        }
                        in_run = FALSE;
                   }
                   /*
                    * Not efficient, get rid of it..
                    */
                   if (rgi->rrgi_Next == NULL)
                       rgi_tail = rgi;
              }
#else
          rgi = r_growth_list;
          rgi_tail = rgi;
          while (rgi != NULL && !r_union_test_table[rgi->rrgi_StateData])
               rgi = rgi->rrgi_Next;
          while (rgi != NULL)
          {
               if (!r_check_rgn_buf_len(dest_size + 2))
                   return FALSE;
               r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
               do
               {
                    rgi = rgi->rrgi_Next;
               } while (rgi != NULL && r_union_test_table[rgi->rrgi_StateData]);
               rgi_tail = rgi;
               r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
               do
               {
                    rgi = rgi->rrgi_Next;
               } while (rgi != NULL && !r_union_test_table[rgi->rrgi_StateData]);
          }
#endif
               if (r_RgnBuf[dest_size - 2] == R_NEXT_IS_Y)
               {
                    /*
                     * We didn't output anything for these input rows. Rewind..
                     */
                    dest_size -= 2;
               }
          }
          /*
           * Now, we've completed using the growth list for constructing this
           * region. Therefore, we add it to the front of the free list, to
```

```
             * be re-used later.
             */
#ifdef R_NEW_IMP_CONSTRUCTION_LOOP
        while (rgi_tail->rrgi_Next != NULL)
             rgi_tail = rgi_tail->rrgi_Next;
#endif
            rgi_tail->rrgi_Next = r_free_list;
            r_free_list  = r_growth_list;
            r_growth_list = NULL;
            /*
             * We've completed constructing the data for the region. We
             * make a copy the constructed data from the permanent buffer to
             * an exactly fitting buffer.
             */
            rgn->rr_RgnData = (R_Int *)malloc(++dest_size * sizeof(R_Int));
            if (rgn->rr_RgnData == NULL)
            {
                return FALSE;
            }
            memcpy(rgn->rr_RgnData, r_RgnBuf, (dest_size - 1) * sizeof(R_Int));
            rgn->rr_RgnData[dest_size - 1] = R_EOR;
            rgn->rr_RgnDataSize = dest_size;
            ASSERT(rgn->rr_RgnDataSize >= 9);
        }
        /*
         * We now do a bounding box union of the two component bboxes and place
         * the result in the new region.
         */
        BB_union(&r1->rr_BBox, &r2->rr_BBox, &rgn->rr_BBox);
        /*
         * Done! We can get out..
         */
        return TRUE;
}

/*
 * R_union_equals
 *
 * This function basically implements a r1 union= r2 type operation. Ie
 * r1 union r2 is calculated and the result returned in r1.
 *
 * Parameters:
 *          r1          A pointer to an R_Region. This represents
 *                      the first half of the union, and is also used to return
 *                      the eventual result.
 *          r2          A pointer to an R_Region. This represents the second
 *                      half of the union.
 * Returns:
 *          TRUE on success, FALSE on failure.
 */
int
R_union_equals
(
        R_Region   *r1,
        R_Region   *r2
)
{
        R_Region   new_rgn;
        If (r1->rr_RgnData == NULL)
            return R_init_region_with_region(r1, r2);
        if (!R_union(&new_rgn, r1, r2))
            return FALSE;
        R_empty_region(r1);
        *r1 = new_rgn;
        return TRUE;
}

/*
```

```
 * r_intersection_test_table
 *
 * A 16-int lookup table which when provided with an unsigned char
 * of the following form xxyy, will provide evaluate the key
 * state transition test of the intersection construction loop.
 * Note that R_STATE_SIZE _must_ be 2 for this lookup table to
 * work.
 */
int r_intersection_test_table[16] = {
                                          0, 0, 0, 1,
                                          0, 0, 0, 1,
                                          0, 0, 0, 1,
                                          1, 1, 1, 0
                                     };


/*
 * R_intersection
 *
 * This function inits a R_Region structure to represent the intersection
 * of it's two arguments.
 *
 * Parameters:
 *       rgn       A R_Region ptr to the R_Region structure to be initialised.
 *       r1        A R_Region ptr representing the first region.
 *       r2        A R_Region ptr representing the second region.
 * Returns
 *       TRUE on success, FALSE on failure.
 */
int
R_intersection
(
     R_Region   *rgn,
     R_Region   *r1,
     R_Region   *r2
)
{
     R_Int      *r1_dat;
     R_Int      *r2_dat;
     int        overlap_flags;


     int_tot++;

     rgn->rr_RgnData = NULL;
     if (!BB_intersect_test(&r1->rr_BBox, &r2->rr_BBox, &overlap_flags))
     {
          /*
           * The bounding boxes don't intersect. This means that the regions
           * don't intersect. Therefore, we simply set rgn->rr_RgnData to NULL
           * (signifying an empty region) and get out..
           */
          return TRUE;
     }
     R_Int                min_row;
     int                  dest_size;
     R_RgnGrowItem        *rgi;
     R_RgnGrowItem        *rgi_tail;
     int                  in_run;
     int                  done_r1_in_row;
     IntXYMinMax          new_bbox;

     int_full++;

     /*
      * The two regions _do_ overlap in x _and y. We therefore have
      * to do a bit more work in calculating the intersection of the two
      * regions. We use the R_RegionBuilder struct to store state
      * regarding the currently active regions as we progress through
 * the rows of each region. After any rows relevent to a y-coord
 * are added to the region builder, we examine the state of each
 * pixel run in the region builder. If the addition of the row(s)
 * for the y-coord have caused a transition to or from 0x3, then
 * the pixel run is emitted.
 */
/*
 * Initialise the new_bbox structure for determining the new bounding box.
 */
new_bbox.X.Min = R_INT_MAX_VALUE;
new_bbox.Y.Min = R_INT_MAX_VALUE;
new_bbox.X.Max = R_INT_MIN_VALUE;
new_bbox.Y.Max = R_INT_MIN_VALUE;

/*
 * The next thing we do is ensure the current region builder is empty,
 * and set up pointers into the region data of the two regions.
 */
r1_dat = r1->rr_RgnData;
r2_dat = r2->rr_RgnData;
dest_size = 0;
/*
 * We are now ready to loop through the data from both regions. Notice
 * that we only keep looping whilst _both_ regions have some data left
 * to give. As soon as either of the region's data has been exhausted,
 * then we stop as the intersection region has already been calculated
 * and is sitting in the rgn_buf.
 */
while (*r1_dat != R_EOR && *r2_dat != R_EOR)
{
     ASSERT(*r1_dat == R_NEXT_IS_Y || *r1_dat == R_EOR);
     ASSERT(*r2_dat == R_NEXT_IS_Y || *r2_dat == R_EOR);
     If  (*r1_dat == R_EOR)
          min_row = r2_dat[1];
     else if (*r2_dat == R_EOR)
          min_row = r1_dat[1];
     else
          min_row = min(r1_dat[1], r2_dat[1]);
     done_r1_in_row = FALSE;
     if (*r1_dat != R_EOR && r1_dat[1] == min_row)
     {
          /*
           * The first region is active on this y coord. We add this
           * row to the current region builder.
           */
          if (!R_add_row_to_region_growth_list(&r1_dat, 0x1, TRUE))
              return FALSE;
          done_r1_in_row = TRUE;
     }
     if (*r2_dat != R_EOR && r2_dat[1] == min_row)
     {
          /*
           * The first region is active on this y coord. We add this
           * row to the current region builder.
           */
          if (!R_add_row_to_region_growth_list(&r2_dat, 0x2, !done_r1_in_row))
              return FALSE;
     }
     /*
      * Now, we generate the output row for the input rows.
      */
     if (!r_check_rgn_buf_len(dest_size + 2))
     {
         return FALSE;
     }
     r_RgnBuf[dest_size++] = R_NEXT_IS_Y;
     r_RgnBuf[dest_size++] = min_row;
     in_run = FALSE;
```

```
#ifndef R_NEW_IMP_CONSTRUCTION_LOOP
          for (rgi = r_growth_list; rgi != NULL; rgi = rgi->rrgi_Next)
          {
#if 0
               if
               (
                    rgi->rrgi_StateData != (3 | (3 << RB_STATE_SIZE))
                    &&
                    (
                         (rgi->rrgi_StateData & RB_PREV_STATE_MASK) == 3
                         ||
                         (rgi->rrgi_StateData & RB_CUR_STATE_MASK) ==
                                                       (3 << RB_STATE_SIZE)
                    )
               )
#else
               if (r_intersection_test_table[rgi->rrgi_StateData])
#endif
               {
                    /*
                     * We have to emit a run here, if we're not already
                     * in one..
                     */
                    if (!in_run)
                    {
                         if (!r_check_rgn_buf_len(dest_size + 1))
                         {
                              return FALSE;
                         }
                         r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
                         in_run = TRUE;
                         new_bbox.X.Min = min(new_bbox.X.Min, rgi->rrgi_RgnData);
                    }
               }
               else
               {
                    if   (in_run)
                    {
                         /*
                          * We've come to the end of a run. We output the next element
to end it.
                          */
                         if (!r_check_rgn_buf_len(dest_size + 1))
                         {
                              return FALSE;
                         }
                         r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
                         new_bbox.X.Max = max(new_bbox.X.Max, rgi->rrgi_RgnData);
                    }
                    in_run = FALSE;
               }
               /*
                * Not efficient, get rid of it..
                */
               if (rgi->rrgi_Next == NULL)
                   rgi_tail = rgi;
          }
#else
          rgi = r_growth_list;
          rgi_tail = rgi;
          while (rgi != NULL && !r_intersection_test_table[rgi->rrgi_StateData])
               rgi = rgi->rrgi_Next;
          while (rgi != NULL)
          {
               if (!r_check_rgn_buf_len(dest_size + 2))
                   return FALSE;
               r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
               new_bbox.X.Min = min(new_bbox.X.Min, rgi->rrgi_RgnData);
```

```
                do
                {
                     rgi = rgi->rrgi_Next;
                } while (rgi != NULL && r_intersection_test_table[rgi->rrgi_StateData]);
                rgi_tail = rgi;
                r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
                new_bbox.X.Max = max(new_bbox.X.Max, rgi->rrgi_RgnData);
                do
                {
                     rgi = rgi->rrgi_Next;
                } while (rgi != NULL && !r_intersection_test_table[rgi-
>rrgi_StateData]);
           }
#endif
           if   (r_RgnBuf[dest_size - 2] == R_NEXT_IS_Y)
           {
                /*
                 * We didn't output anything for these input rows. Rewind..
                 */
                dest_size -= 2;
           }
           else
           {
                if (min_row < new_bbox.Y.Min)
                    new_bbox.Y.Min = min_row;
                else if (min_row > new_bbox.Y.Max)
                    new_bbox.Y.Max = min_row;
           }
      }
      /*
       * Now, we've completed using the growth list for constructing this
       * region. Therefore, we add it to the front of the free list, to
       * be re-used later.
       */
#ifdef R_NEW_IMP_CONSTRUCTION_LOOP
       while (rgi_tail->rrgi_Next != NULL)
            rgi_tail = rgi_tail->rrgi_Next;
#endif
      rgi_tail->rrgi_Next = r_free_list;
      r_free_list  = r_growth_list;
      r_growth_list = NULL;
      /*
       * We've completed constructing the data for the region. Firstly
       * we check to see if we've emitted anything at all. If we have
       * then dest_size must be > 0. If it isn't we simply free the
       * region we created and get out, as the regions don't really
       * intersect, in spite of their intersecting bounding boxes.
       */
      if (dest_size == 0)
      {
           return TRUE;
      }
      /*
       * We make a copy the constructed data from the permanent buffer to
       * an exactly fitting buffer.
       */
      rgn->rr_RgnData = (R_Int *)malloc(++dest_size * sizeof(R_Int));
      if (rgn->rr_RgnData == NULL)
      {
           return FALSE;
      }
      memcpy(rgn->rr_RgnData, r_RgnBuf, (dest_size - 1) * sizeof(R_Int));
      rgn->rr_RgnData[dest_size - 1] = R_EOR;
      rgn->rr_RgnDataSize = dest_size;
      ASSERT(rgn->rr_RgnDataSize >= 9);
      /*
       * Now, copy across the bounding box.. Before we do this, we subtract
       * 1 from X.Max and Y.Max because of the region format.
```

```
         */
        new_bbox.X.Max--;
        new_bbox.Y.Max--;
        rgn->rr_BBox = new_bbox;
        /*
         * Done! We can get out..
         */
        return TRUE;
}

/*
 * r_difference_test_table
 *
 * A 16-int lookup table which when provided with an unsigned char
 * of the following form xxyy, will provide evaluate the key
 * state transition test of the difference construction loop.
 * Note that R_STATE_SIZE _must_ be 2 for this lookup table to
 * work.
 */
int r_difference_test_table[16] = {
                                        0, 1, 0, 0,
                                        1, 0, 1, 1,
                                        0, 1, 0, 0,
                                        0, 1, 0, 0
                                    };

/*
 * R_difference
 *
 * This function inits a R_Region structure to represent the difference of
 * it's two arguments. It essentially calculates r1 - r2
 *
 * Parameters:
 *          rgn         A R_Region ptr representing the R_Region to be inited.
 *          r1          A R_Region ptr representing the first region.
 *          r2          A R_Region ptr representing the second region.
 * Returns
 *          TRUE on success, FALSE on failure.
 */
int
R_difference
(
        R_Region    *rgn,
        R_Region    *r1,
        R_Region    *r2
)
{
        R_Int       *r1_dat;
        R_Int       *r2_dat;
        int         overlap_flags;


        diff_tot++;

        rgn->rr_RgnData = NULL;
        if (!BB_intersect_test(&r1->rr_BBox, &r2->rr_BBox, &overlap_flags))
        {
                /*
                 * The bounding boxes don't intersect. This means that r1 - r2
                 * simply equals r1. We make a copy of the relevant bits and get out..
                 */
                rgn->rr_BBox = r1->rr_BBox;
                rgn->rr_RgnDataSize = r1->rr_RgnDataSize;
                rgn->rr_RgnData = (R_Int *)malloc(r1->rr_RgnDataSize * sizeof(R_Int));
                if (rgn->rr_RgnData == NULL)
                {
                        return FALSE;
                }
```

```
      memcpy
      (
           rgn->rr_RgnData,
           r1->rr_RgnData,
           r1->rr_RgnDataSize * sizeof(R_Int)
      );
      return TRUE;
}
R_Int           min_row;
int             dest_size;
R_RgnGrowItem   *rgi;
R_RgnGrowItem   *rgi_tail;
int             in_run;
int             done_r1_in_row;
unsigned char   m_high;
unsigned char   m_low;
IntXYMinMax     new_bbox;

diff_full++;

/*
 * The two regions _do_ overlap in x _and y. We therefore have
 * to do a bit more work in calculating the difference of the two
 * regions. We use the R_RegionBuilder struct to store state
 * regarding the currently active regions as we progress through
 * the rows of each region. After any rows relevent to a y-coord
 * are added to the region builder, we examine the state of each
 * pixel run in the region builder. If the addition of the row(s)
 * for the y-coord have caused the following transitions -
 *              r1      -> 0
 *              0       -> r2
 *              r1 + r2 -> r2
 *              r2      -> r1 + r2
 * ..then the relevent runs are emitted. Firstly, though,
 * we ensure the current region builder is empty,
 * and set up pointers into the region data of the two regions.
 */
r1_dat = r1->rr_RgnData;
r2_dat = r2->rr_RgnData;
dest_size = 0;

/*
 * Initialise the new_bbox structure for determining the new bounding box.
 */
new_bbox.X.Min = 32767;
new_bbox.Y.Min = 32767;
new_bbox.X.Max = -32768;
new_bbox.Y.Max = -32768;

/*
 * We are now ready to loop through the data from both regions. Notice
 * that we only keep looping whilst r1 has data outstanding. When
 * r1's data is consumed, then any transitions made by r2 are
 * irrelevant.
 */
while (*r1_dat != R_EOR)
{
      ASSERT(*r1_dat == R_NEXT_IS_Y || *r1_dat == R_EOR);
      ASSERT(*r2_dat == R_NEXT_IS_Y || *r2_dat == R_EOR);
      if (*r1_dat == R_EOR)
          min_row = r2_dat[1];
      else if (*r2_dat == R_EOR)
          min_row = r1_dat[1];
      else
          min_row = min(r1_dat[1], r2_dat[1]);
      done_r1_in_row = FALSE;
      if (*r1_dat != R_EOR && r1_dat[1] == min_row)
      {
```

```
                /*
                 * The first region is active on this y coord. We add this
                 * row to the current region builder.
                 */
                if (!R_add_row_to_region_growth_list(&r1_dat, 0x1, TRUE))
                    return FALSE;
                done_r1_in_row = TRUE;
            }
            if (*r2_dat != R_EOR && r2_dat[1] == min_row)
            {
                /*
                 * The first region is active on this y coord. We add this
                 * row to the current region builder.
                 */
                if (!R_add_row_to_region_growth_list(&r2_dat, 0x2, !done_r1_in_row))
                    return FALSE;
            }
            /*
             * Now, we generate the output row for the input rows.
             */
            if (!r_check_rgn_buf_len(dest_size + 2))
            {
                return FALSE;
            }
            r_RgnBuf[dest_size++] = R_NEXT_IS_Y;
            r_RgnBuf[dest_size++] = min_row;
            in_run = FALSE;
#ifndef R_NEW_IMP_CONSTRUCTION_LOOP
            for (rgi = r_growth_list; rgi != NULL; rgi = rgi->rrgi_Next)
            {
#if 0
                m_high = (rgi->rrgi_StateData & RB_CUR_STATE_MASK) >> RB_STATE_SIZE;
                m_low = rgi->rrgi_StateData & RB_PREV_STATE_MASK;
                if
                (
                    (
                        (m_low != 1 && m_high == 1)
                        ||
                        (m_low == 1 && m_high != 1)
                    )
                )
#else
                if    (r_difference_test_table[rgi->rrgi_StateData])
#endif
                {
                    /*
                     * We have to emit a run here, if we're not already
                     * in one..
                     */
                    if (!in_run)
                    {
                        if (!r_check_rgn_buf_len(dest_size + 1))
                        {
                            return FALSE;
                        }
                        r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
                        in_run = TRUE;
                        new_bbox.X.Min = min(new_bbox.X.Min, rgi->rrgi_RgnData);
                    }
                }
                else
                {
                    if (in_run)
                    {
                        /*
                         * We've come to the end of a run. We output the next element
                         to end it.
                         */
                        if (!r_check_rgn_buf_len(dest_size + 1))
                        {
                            return FALSE;
                        }
                        r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
                        new_bbox.X.Max = max(new_bbox.X.Max, rgi->rrgi_RgnData);
                    }
                    in_run = FALSE;
                }
                /*
                 * Not efficient, get rid of it..
                 */
                if (rgi->rrgi_Next == NULL)
                    rgi_tail = rgi;
            }
#else
            rgi = r_growth_list;
            rgi_tail = rgi;
            while (rgi != NULL && !r_difference_test_table[rgi->rrgi_StateData])
                rgi = rgi->rrgi_Next;
            while (rgi != NULL)
            {
                if (!r_check_rgn_buf_len(dest_size + 2))
                    return FALSE;
                r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
                new_bbox.X.Min = min(new_bbox.X.Min, rgi->rrgi_RgnData);
                do
                {
                    rgi = rgi->rrgi_Next;
                } while (rgi != NULL && r_difference_test_table[rgi->rrgi_StateData]);
                rgi_tail = rgi;
                r_RgnBuf[dest_size++] = rgi->rrgi_RgnData;
                new_bbox.X.Max = max(new_bbox.X.Max, rgi->rrgi_RgnData);
                do
                {
                    rgi = rgi->rrgi_Next;
                } while (rgi != NULL && !r_difference_test_table[rgi->rrgi_StateData]);
            }
#endif
            if    (r_RgnBuf[dest_size - 2] == R_NEXT_IS_Y)
            {
                /*
                 * We didn't output anything for these input rows. Rewind..
                 */
                dest_size -= 2;
            }
            else
            {
                if (min_row < new_bbox.Y.Min)
                    new_bbox.Y.Min = min_row;
                else if (min_row > new_bbox.Y.Max)
                    new_bbox.Y.Max = min_row;
            }
        }
        /*
         * Now, we've completed using the growth list for constructing this
         * region. Therefore, we add it to the front of the free list, to
         * be re-used later.
         */
#ifdef R_NEW_IMP_CONSTRUCTION_LOOP
        while (rgi_tail->rrgi_Next != NULL)
            rgi_tail = rgi_tail->rrgi_Next;
#endif
        rgi_tail->rrgi_Next = r_free_list;
        r_free_list  = r_growth_list;
        r_growth_list = NULL;
        /*
         * We've completed constructing the data for the region. Firstly
         * we check to see if we've emitted anything at all. If we have
         * then dest_size must be > 0. If it isn't we simply free the
         * region we created and get out, as r2 - r1 must be empty.
         */
        if (dest_size == 0)
        {
            return TRUE;
        }
        /*
         * We make a copy the constructed data from the permanent buffer to
         * an exactly fitting buffer.
         */
        rgn->rr_RgnData = (R_Int *)malloc(++dest_size * sizeof(R_Int));
        if (rgn->rr_RgnData == NULL)
        {
            return FALSE;
        }
        memcpy(rgn->rr_RgnData, r_RgnBuf, (dest_size - 1) * sizeof(R_Int));
        rgn->rr_RgnData[dest_size - 1] = R_EOR;
        rgn->rr_RgnDataSize = dest_size;
        ASSERT(rgn->rr_RgnDataSize >= 9);
        /*
         * Now, copy across the bounding box..
         */
        rgn->rr_BBox = new_bbox;
        /*
         * Done! We can get out..
         */
        return TRUE;
}
#endif /* R_USE_NEW_IMP */

/*
 * R_compare
 *
 * This function compares two regions and determines if they are the same.
 *
 * Parameters:
 *          rgn1    The first R_Region.
 *          rgn2    The second R_Region.
 * Returns:
 *          TRUE if they are the same, FALSE if they aren't.
 */
int
R_compare
(
        R_Region        *rgn1,
        R_Region        *rgn2
)
{
        /*
         * If their region data sizes don't agree, then they aren't the same.
         */
        if    (rgn1->rr_RgnDataSize != rgn2->rr_RgnDataSize)
               return FALSE;
        if
        (
              memcmp
              (
                    rgn1->rr_RgnData,
                    rgn2->rr_RgnData,
                    rgn1->rr_RgnDataSize * sizeof(R_Int)
              )
              ==
              0
        )
              return TRUE;
        return FALSE;
```

```
}

/*
 * r_check_rect_buf_len
 *
 * This function checks to see if the static rectangle buffer is large
 * enough. If it isn't then it is reallocated to make it large enough.
 *
 * Parameters:
 *         size The required size of the r_RectBuf array.
 * Returns:
 *         TRUE on success, FALSE on failure.
 */
static int
r_check_rect_buf_len
(
     int        size
)
{
     ASSERT(size >= 0);
     if (size > r_RectBufSize)
     {
          int              new_buf_size;
          IntXYMinMax      *new_buf;
          new_buf_size = max(size, r_RectBufSize * 2);
          new_buf = (IntXYMinMax *)malloc
                          (new_buf_size * sizeof(IntXYMinMax)
                     );
          if (new_buf == NULL)
               return FALSE;
          if (r_RectBuf != NULL)
          {
               memcpy(new_buf, r_RectBuf, r_RectBufSize * sizeof(IntXYMinMax));
               free(r_RectBuf);
          }
          r_RectBuf = new_buf;
          r_RectBufSize = new_buf_size;
     }
     return TRUE;
}

#ifndef R_USE_NEW_IMP
/*
 * R_rects_from_region
 *
 * This function returns a group of non-overlapping rectangles which
 * together constitute the region. The group of rectangles returned is
 * currently non-optimal as the function uses the R_RegionBuilder structure
 * to store state. A more specific data structure will be required to
 * make the rectangles produced more optimal.
 *
 * Parameters:
 *         rgn       The region from which a rectangle array is required.
 *         rects     A pointer to a pointer to a IntXYMinMax structure. Used
 *                   to return the array.
 *         num_rects A pointer to an int. Used to return the number of
 *                   elements in the array.
 *         static_ok This boolean arg is passed as TRUE if a pointer to
 *                   the r_RectBuf is sufficient. This is TRUE if usefulness of
 *                   the rectangle data obtained ends before the next call to
 *                   R_rects_from_region (for any region). FALSE is passed if
 *                   a newly malloced copy is required. Basically is TRUE is
 *                   passed the pointer returned must _not_ be freed.
 * Returns:
 *         TRUE on success, FALSE on failure.
 */
int
R_rects_from_region
```

```
(
     R_Region         *rgn,
     IntXYMinMax      **rects,
     int              *num_rects,
     int              static_ok
)
{
     R_Int                  *rgn_data;
     int                    dest_index;
     int                    prev_y;
     int                    prev_x;
     unsigned char          *rgn_bld_stat;
     R_Int                  *rgn_bld_dat;
     int                    i;
     int                    in_run;

     /*
      * Give "nice" defaults for return stuff in cause we fail..
      */
     *rects = NULL;
     *num_rects = 0;
     /*
      * We grab a pointer to the region data for the region and ensure
      * that the current region builder is empty..
      */
     rgn_data = rgn->rr_RgnData;
     if   (rgn_data == NULL)
          /*
           * This is an empty region.. Get out..
           */
           return TRUE;
     ASSERT(*rgn_data == R_NEXT_IS_Y);
     R_CurRB->rrb_Nels = 0;
     /*
      * We add the first row of the region to the region builder. We also
      * store the y-coord of this first row.
      */
     prev_y = rgn_data[1];
     if (!R_add_row_to_region_builder(&rgn_data, 0x1, TRUE))
          return FALSE;
     ASSERT(*rgn_data == R_NEXT_IS_Y);
     ASSERT(*rgn_data != R_EOR);
     /*
      * We are now in a position to loop through the data of the region.
      * We continue until the region data runs out. Basically, we output
      * the runs in the current region builder out as rectangles. Using
      * x-coords from the region builder and y coords of the rows. Then,
      * we add then next row to the region builder.
      */
     dest_index = 0;
     while (*rgn_data != R_EOR)
     {
          ASSERT(*rgn_data == R_NEXT_IS_Y);

          rgn_bld_stat = R_CurRB->rrb_StateData;
          rgn_bld_dat = R_CurRB->rrb_RgnData;
          in_run = FALSE;
          for (i = R_CurRB->rrb_Nels; i > 0; i--)
          {
               if ((*rgn_bld_stat & RB_CUR_STATE_MASK) > 0)
               {
                    /*
                     * We have to emit a run here, if we're not already
                     * in one..
                     */
                    if (!in_run)
                    {
                         prev_x = *rgn_bld_dat;
```

```
                        in_run = TRUE;
                    }
                }
                else
                {
                    if (in_run)
                    {
                        /*
                         * We've come to the end of a run. We output the rectangle
                         * right here..
                         */
                        if (!r_check_rect_buf_len(dest_index + 1))
                            return FALSE;
                        r_RectBuf[dest_index].X.Min = prev_x;
                        r_RectBuf[dest_index].Y.Min = prev_y;
                        r_RectBuf[dest_index].X.Max = *rgn_bld_dat - 1;
                        r_RectBuf[dest_index++].Y.Max = rgn_data[1] - 1;
                    }
                    in_run = FALSE;
                }
                rgn_bld_stat++;
                rgn_bld_dat++;
            }
            /*
             * Now, we advance onto the next row..
             */
            prev_y = rgn_data[1];
            if (!R_add_row_to_region_builder(&rgn_data, 0x1, TRUE))
                return FALSE;
        }
        /*
         * Ok, we have the array of rectangles sitting around. If static_ok
         * is TRUE then we simply set the return pointers and get out.
         * Otherwise, we need to malloc a copy of the r_RectBuf.
         */
        *num_rects = dest_index;
        if (static_ok)
        {
            *rects = r_RectBuf;
        }
        else
        {
            *rects = (IntXYMinMax *)malloc(dest_index * sizeof(IntXYMinMax));
            if (*rects == NULL)
                return FALSE;
            memcpy(*rects, r_RectBuf, dest_index * sizeof(IntXYMinMax));
        }
        return TRUE;
}
#else
/*
 * R_rects_from_region
 *
 * This function returns a group of non-overlapping rectangles which
 * together constitute the region. The group of rectangles returned is
 * currently non-optimal as the function uses the R_RegionBuilder structure
 * to store state. A more specific data structure will be required to
 * make the rectangles produced more optimal.
 *
 * Parameters:
 *          rgn        The region from which a rectangle array is required.
 *          rects      A pointer to a pointer to a IntXYMinMax structure. Used
 *                     to return the array.
 *          num_rects  A pointer to an int. Used to return the number of
 *                     elements in the array.
 *          static_ok  This boolean arg is passed as TRUE if a pointer to
 *                     the r_RectBuf is sufficient. This is TRUE if usefulness of
 *                     the rectangle data obtained ends before the next call to
 *                      R_rects_from_region (for any region). FALSE is passed if
 *                      a newly malloced copy is required. Basically is TRUE is
 *                      passed the pointer returned must _not_ be freed.
 * Returns:
 *          TRUE on success, FALSE on failure.
 */
int
R_rects_from_region
(
     R_Region          *rgn,
     IntXYMinMax       **rects,
     int               *num_rects,
     int               static_ok
)
{
     R_Int                  *rgn_data;
     int                    dest_index;
     R_RgnGrowItem          *rgi;
     R_RgnGrowItem          *rgi_tail;
     int                    prev_y;
     int                    prev_x;
     int                    in_run;

     /*
      * Give "nice" defaults for return stuff in cause we fail..
      */
     *rects = NULL;
     *num_rects = 0;
     /*
      * We grab a pointer to the region data for the region and ensure
      * that the current region builder is empty..
      */
     rgn_data = rgn->rr_RgnData;
     if (rgn_data == NULL)
          /*
           * This is an empty region.. Get out..
           */
           return TRUE;
     ASSERT(*rgn_data == R_NEXT_IS_Y);
     /*
      * We add the first row of the region to the region builder. We also
      * store the y-coord of this first row.
      */
     prev_y = rgn_data[1];
     if (!R_add_row_to_region_growth_list(&rgn_data, 0x1, TRUE))
         return FALSE;
     ASSERT(*rgn_data == R_NEXT_IS_Y);
     ASSERT(*rgn_data != R_EOR);
     /*
      * We are now in a position to loop through the data of the region.
      * We continue until the region data runs out. Basically, we output
      * the runs in the current region builder out as rectangles. Using
      * x-coords from the region builder and y coords of the rows. Then,
      * we add then next row to the region builder.
      */
     dest_index = 0;
     while (*rgn_data != R_EOR)
     {
          ASSERT(*rgn_data == R_NEXT_IS_Y);

          in_run = FALSE;
          for (rgi = r_growth_list; rgi != NULL; rgi = rgi->rrgi_Next)
          {
               if ((rgi->rrgi_StateData & RB_CUR_STATE_MASK) > 0)
               {
                    /*
                     * We have to emit a run here, if we're not already
                     * in one..
```

```
                         */
                        if (!in_run)
                        {
                             prev_x = rgi->rrgi_RgnData;
                             in_run = TRUE;
                        }
                   }
                   else
                   {
                        if (in_run)
                        {
                             /*
                              * We've come to the end of a run. We output the rectangle
                              * right here..
                              */
                             if (!r_check_rect_buf_len(dest_index + 1))
                                 return FALSE;
                             r_RectBuf[dest_index].X.Min = prev_x;
                             r_RectBuf[dest_index].Y.Min = prev_y;
                             r_RectBuf[dest_index].X.Max = rgi->rrgi_RgnData - 1;
                             r_RectBuf[dest_index++].Y.Max = rgn_data[1] - 1;
                        }
                        in_run = FALSE;
                   }
                   /*
                    * Not efficient, get rid of it..
                    */
                   if (rgi->rrgi_Next == NULL)
                       rgi_tail = rgi;
              }
              /*
               * Now, we advance onto the next row..
               */
              prev_y = rgn_data[1];
              if (!R_add_row_to_region_growth_list(&rgn_data, 0x1, TRUE))
                  return FALSE;
         }
         /*
          * Now, we've completed using the growth list for constructing the
          * rect list. Therefore, we add it to the front of the free list, to
          * be re-used later.
          */
         rgi_tail->rrgi_Next = r_free_list;
         r_free_list  = r_growth_list;
         r_growth_list = NULL;
         /*
          * Ok, we have the array of rectangles sitting around. If static_ok
          * is TRUE then we simply set the return pointers and get out.
          * Otherwise, we need to malloc a copy of the r_RectBuf.
          */
         *num_rects = dest_index;
         if (static_ok)
         {
              *rects = r_RectBuf;
         }
         else
         {
              *rects = (IntXYMinMax *)malloc(dest_index * sizeof(IntXYMinMax));
              if (*rects == NULL)
                  return FALSE;
              memcpy(*rects, r_RectBuf, dest_index * sizeof(IntXYMinMax));
         }
         return TRUE;
    }
    #endif /* R_USE_NEW_IMP */

    /*
     * R_translate_region
```

```
 *
 * This function simply translates a region by the delta provided.
 *
 * Parameters:
 *      rgn     A ptr to the R_Region to be translated.
 *      delta   An IntXY ptr representing the amount to translate
 *              in x and y.
 * Returns:
 *      Nothing.
 */
void
R_translate_region
(
    R_Region    *rgn,
    IntXY       *delta
)
{
    R_Int   *rgn_data;
    BB_translate(&rgn->rr_BBox, delta);
    rgn_data = rgn->rr_RgnData;
    for (int i = 0; i < rgn->rr_RgnDataSize - 1; i++)
    {
        if (rgn_data[i] == R_NEXT_IS_Y)
        {
            i++;
            rgn_data[i] += + delta->Y;
            continue;
        }
        rgn_data[i] += delta->X;
    }
}

/*
 * R_output_region_as_debug_string
 *
 * This function simply outputs a region's data using the debug string
 * functionality.
 *
 * Parameters:
 *         rgn_name    A string used to output a user-defined name for the
 *                     region.
 *         rgn         The region to be output.
 * Returns:
 *         Nothing.
 */
void
R_output_region_as_debug_string
(
     char       *rgn_name,
     R_Region   *rgn
)
{
     char buffer[128];
     int        index;
     int        line_len;

     sprintf(buffer, "\n+Rgn : %s\n", rgn_name);
     OutputDebugString(buffer);
     if (rgn == NULL)
     {
          sprintf(buffer, "+----End %s (Empty)\n", rgn_name);
          OutputDebugString(buffer);
          return;
     }
     sprintf
     (
          buffer,
          "+----BBox:(%d, %d, %d, %d)\n",
            rgn->rr_BBox.X.Min,
            rgn->rr_BBox.Y.Min,
            rgn->rr_BBox.X.Max,
            rgn->rr_BBox.Y.Max
        );
        OutputDebugString(buffer);
        sprintf(buffer, "+----Nels: %d\n", rgn->rr_RgnDataSize);
        OutputDebugString(buffer);
        sprintf(buffer, "+----Data: ...");
        OutputDebugString(buffer);
        for (index = 0; index < rgn->rr_RgnDataSize; index++)
        {
            if (rgn->rr_RgnData[index] == R_NEXT_IS_Y)
            {
                sprintf(buffer, "\n|           Y:%3d--> ", rgn->rr_RgnData[++index]);
                line_len = strlen(buffer);
                OutputDebugString(buffer);
            }
            else if (rgn->rr_RgnData[index] == R_EOR)
            {
                sprintf(buffer, "\n+----End %s\n", rgn_name);
                OutputDebugString(buffer);
            }
            else
            {
                sprintf(buffer, "%3d, ", rgn->rr_RgnData[index]);
                if (strlen(buffer) + line_len > 80)
                {
                    OutputDebugString("\n|                      ");
                    line_len = strlen("\n|                      ");
                }
                OutputDebugString(buffer);
                line_len += strlen(buffer);
            }
        }
}

#define NUM_ITERATIONS    200

int
R_test_new_region_arithmetic()
{
        R_Region        rgn1;
        R_Region        rgn2;
        R_Region        rgn3;
        R_Region        rgn4;
        R_Region        rgn5;
        R_Region        rgn6;
        IntXYMinMax     rect;
        int             i;
        IntXY           delta;
        char            buf[256];
        unsigned long   ticks_new;
        unsigned long   ticks_old;

#if 0
        /*
         * Union Test.
         */
        ticks_new = GetTickCount();
        rect.X.Min = 50;
        rect.Y.Min = 50;
        rect.X.Max = 100;
        rect.Y.Max = 100;
        if (!R_init_region_with_rect(&rgn1, &rect))
            return FALSE;
        rect.X.Min = 70;
        rect.Y.Min = 70;
rect.X.Max = 120;
rect.Y.Max = 120;
if (!R_init_region_with_rect(&rgn2, &rect))
    return FALSE;
if (!R_union_list_equals(&rgn1, &rgn2))
    return FALSE;
delta.X = 5;
delta.Y = 5;
for (i = 0; i < NUM_ITERATIONS; i++)
{
     R_translate_region(&rgn2, &delta);
     if (!R_union_list_equals(&rgn1, &rgn2))
         return FALSE;
}
ticks_new = GetTickCount() - ticks_new;

ticks_old = GetTickCount();
rect.X.Min = 50;
rect.Y.Min = 50;
rect.X.Max = 100;
rect.Y.Max = 100;
if (!R_init_region_with_rect(&rgn3, &rect))
    return FALSE;
rect.X.Min = 70;
rect.Y.Min = 70;
rect.X.Max = 120;
rect.Y.Max = 120;
if (!R_init_region_with_rect(&rgn4, &rect))
    return FALSE;
if (!R_union_equals(&rgn3, &rgn4))
    return FALSE;
delta.X = 5;
delta.Y = 5;
for (i = 0; i < NUM_ITERATIONS; i++)
{
     R_translate_region(&rgn4, &delta);
     if (!R_union_equals(&rgn3, &rgn4))
         return FALSE;
}
ticks_old = GetTickCount() - ticks_old;

if (R_compare(&rgn1, &rgn3))
    sprintf(buf, "New & Old Region Implementations match.\n");
else
    sprintf(buf, "New & Old Region Implementations DO NOT match.\n");
OutputDebugString(buf);
sprintf(buf, "Union Timings - New=%d vs Old=%d\n", ticks_new, ticks_old);
OutputDebugString(buf);
//R_output_region_as_debug_string("New Region Description", &rgn1);
//R_output_region_as_debug_string("Old Region Description", &rgn3);

/*
 * Intersection Test.
 */
R_empty_region(&rgn2);
R_empty_region(&rgn4);
rect.X.Min = 70;
rect.Y.Min = 70;
rect.X.Max = 120;
rect.Y.Max = 120;
if (!R_init_region_with_rect(&rgn2, &rect))
    return FALSE;
delta.X = 5;
delta.Y = 5;
for (i = 0; i < NUM_ITERATIONS; i++)
{
     if (!R_intersection_list(&rgn5, &rgn1, &rgn2))
          return FALSE;
          if (!R_intersection(&rgn6, &rgn3, &rgn2))
              return FALSE;
          if (!R_compare(&rgn5, &rgn6))
          {
              sprintf(buf, "New & Old Region Implementations DO NOT match.\n");
              OutputDebugString(buf);
          }
          R_empty_region(&rgn5);
          R_empty_region(&rgn6);
          R_translate_region(&rgn2, &delta);
     }

     ticks_new = GetTickCount();
     R_empty_region(&rgn2);
     rect.X.Min = 70;
     rect.Y.Min = 70;
     rect.X.Max = 120;
     rect.Y.Max = 120;
     if (!R_init_region_with_rect(&rgn2, &rect))
         return FALSE;
     delta.X = 5;
     delta.Y = 5;
     for (i = 0; i < NUM_ITERATIONS; i++)
     {
          if (!R_intersection_list(&rgn5, &rgn1, &rgn2))
              return FALSE;
          R_empty_region(&rgn5);
          R_translate_region(&rgn2, &delta);
     }
     ticks_new = GetTickCount() - ticks_new;

     ticks_old = GetTickCount();
     R_empty_region(&rgn2);
     rect.X.Min = 70;
     rect.Y.Min = 70;
     rect.X.Max = 120;
     rect.Y.Max = 120;
     if (!R_init_region_with_rect(&rgn2, &rect))
         return FALSE;
     delta.X = 5;
     delta.Y = 5;
     for (i = 0; i < NUM_ITERATIONS; i++)
     {
          if (!R_intersection(&rgn6, &rgn3, &rgn2))
              return FALSE;
          R_empty_region(&rgn6);
          R_translate_region(&rgn2, &delta);
     }
     ticks_old = GetTickCount() - ticks_old;

     sprintf(buf, "Intersection Timings - New=%d vs Old=%d\n", ticks_new, ticks_old);
     OutputDebugString(buf);
     //R_output_region_as_debug_string("New Region Description", &rgn1);
     //R_output_region_as_debug_string("Old Region Description", &rgn3);

     R_empty_region(&rgn1);
     R_empty_region(&rgn2);
     R_empty_region(&rgn3);
     R_empty_region(&rgn4);
     OutputDebugString("Done!!\n");
#endif
     return TRUE;
}
```